On Building Secure Communication Systems

Carvalho Quaresma, Jose Nuno

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

On Building Secure Communication Systems

Jose Quaresma

Kgs Lyngby 2013
PhD-2013-313

DTU Compute
Matematiktorvet, building 303B
DK-2800 Kgs Lyngby
Denmark
Tel +45 4525 3031
Fax +45 4588 1399
compute@compute.dtu.dk
http://www.compute.dtu.dk
PhD-2013-313

Summary

This thesis presents the Guided System Development (GSD) framework, which aims at supporting the development of secure communication systems.

A communication system is specified in a language similar to the Alice and Bob notation, a simple and intuitive language used to describe the global perspective of the communications between different principals. The notation used in the GSD framework extends that notation with constructs that allow the security requirements of the messages to be described.

From that specification, the developer is guided through a semi-automatic translation that enables the verification and implementation of the system. The translation is semi-automatic because the developer has the option of choosing which implementation to use in order to achieve the specified security requirements. The implementation options are given by plugins defined in the framework. The framework's flexibility allows for the addition of constructs that model new security properties as well as new plugins that implement the security properties.

In order to provide higher security assurances, the system specification can be verified by formal methods tools such as the Beliefs and Knowledge (BAK) tool — developed specifically for the GSD framework —, LySatool and OFMC. The framework's flexibility and the existence of the system model in different perspectives — an overall global perspective and an endpoint perspective — allow the connection to new formal methods tools.

The modeled system is also translated into code that implements the communication skeleton of the system and can then be used by the system designer. New output languages can also easily be added to the GSD framework.

Additionally, a prototype of the GSD framework was implemented and an example of using the GSD framework in a real world system is presented.

Resumé

Denne afhandling præsenterer softwareplatformen Guided System Development (GSD), der har til formål at understøtte udviklingen af sikre kommunikationssystemer.

Et kommunikationssystem er specificeret i et sprog der minder om Alice og Bob notationen; et simpelt og intuitivt sprog der beskriver det globale syn på kommunikation mellem forskellige aktører. Notationen brugt på GSD platformen udvider denne notation med begreber der tillader at sikkerhedskrav kan knyttes til udvekslede beskeder.

Ud fra denne specifikation bliver udvikleren ført gennem en delvis automatisk oversættelse, der muliggør verificering og implementering af systemet. Oversættelsen er delvis, fordi udvikleren har mulighed for at vælge hvilken implementation der skal bruges for at opnå de ønskede sikkerhedskrav. Implementationsmulighederne er givet ved moduler defineret til platformen. Platformens fleksibilitet tillader at der bliver tilføjet yderligere begreber der modellerer nye sikkerhedsegenskaber samt nye moduler der implementerer disse egenskaber.

For at kunne give stærkere sikkerhedsgarantier kan specifikationen blive verificeret af forskellige værktøjer inden for formelle metoder, så som Beliefs and Knowledge (BAK) — udviklet specielt til GSD platformen —, LySatool og OFMC. Platformens fleksibilitet — og det at systemet kan modelleres både fra et globalt synspunkt og et endepunkts-synspunkt — tillader at platformen også kan bruges med nye værktøjer.

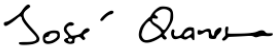
Preface

This thesis was prepared at DTU Compute (formerly DTU Informatics), Technical University of Denmark in fulfillment of the requirements for acquiring a Ph.D. degree in Computer Science.

The Ph.D. study has been carried out under the supervision of Associate Professor Christian W. Probst and Professor Flemming Nielson in the period between September 2010 and August 2013.

Most of the work behind this dissertation has been carried out independently and I take full responsibility for its contents. Some of the work presented is based on articles published in collaboration with my supervisors and the work presented in Chapter 8 was performed in collaboration with Kristin Y. Rozier during my research external stay at NASA Ames Research Center.

Kgs. Lyngby, 31-August,2013



Acknowledgements

I would like to thank my supervisor Christian W. Probst and Flemming Nielson for all their help and support.

I would also like to thank Kristin Y. Rozier, who supervised my external stay at NASA Ames Research Center in Moffett Field, California.

Furthermore, I would like to thank all my colleagues at DTU Compute, and LBT in particular, who helped making my time as a Ph.D. student more pleasant. I am specially thankful to all the ones who shared office with me at some point during my Ph.D. studies and had to put up with all my good (if I might say so myself) craziness.

I would also like to give a special thank you to Sebastian Mödersheim and Roberto Vigo for all the help and fruitful discussions.

I would, of course, also like to thank my family and friends for all their support.

Finally, I would like to thank coffee, without which the writing of this thesis would never have been possible.

Contents

Chapter 1

Introduction

> If you reveal your secrets to the wind, you should not blame the wind for revealing them to the trees.
>
> Kahlil Gibran

Presently, the need for security in the Internet and other communication systems is ever-increasing and, therefore, it is extremely important to provide tools that help system developers to specify communication systems, to choose possible security implementations, to verify the security properties of those systems, and to generate code that implements the system's functionality.

Formal methods provide powerful tools that can be used to verify the security properties of communication systems. However, these tools are not always integrated with the used development environment. This shortcoming is aggravated by the necessary expertise to use formal methods and interpret their results. This obstacle can be overcome by using a framework that aids and guides the developer on the specification of the system under development, on choosing the appropriate implementations that achieve the required security properties, on allowing the automatic verification of its security properties, and also on providing an implementation of the specified system.

That is closely related with the main thesis of my research and, therefore, of the work presented here, which can be summarized as follows:

Security is crucial in communication systems, thus it is important to aid system designers on the implementation of secure communication systems. This work presents a framework that, with the use of step-wise refinement and formal methods, helps a system designer achieving the desired system security without requiring the knowledge to work directly with formal methods tools.

A brief overview of the aforementioned framework, the Guided System Development framework (GSD) is given in Section 1.3.

1.1 Background

In this section, a brief overview of the state-of-the-art of the topics covered in this work is given. A more thorough review is presented in Section 2.

Even though the Service-Oriented Systems (SOS) paradigm is not directly targeted in this work, it has been widely used and researched and, therefore, is worth studying. Service-Oriented Systems are composed by several independent units (principals) that communicate between themselves by exchanging messages. A significant amount of work has been done on formalizing frameworks that have a similar aim to the GSD framework, namely the modeling of SOS and verification of its security properties. One such framework is CaPiTo, presented in Section 2.1.1. Another project with similar goals is AVANTSSAR (Section 2.1.2). More extensive work has been done on the formalization and development of languages that allow the specification and the reasoning of Service-Oriented Systems. In Section 2.3.1, KLAIM, a process calculus to model Service-Oriented Systems, is introduced. Another language targeted at SOS is CaSPiS, presented in Section 2.3.3. Also worth mentioning, specially for the influence that the Alice and Bob notation had in the work presented here, is the AnB language, which is presented in Section 2.3.2.

Security standard suites are generally used in the development of secure communication systems. In the GSD framework, they are used as plugins to implement the different desired security properties. These suites are, for example, WS-Security for SOAP web services, TLS/SSL, or IPSEC. These standards are presented in more detail in Section 5.1.1.

Another important area of research for this work is the use of formal methods. More specifically, the methods and tools targeted at verifying security proper-

ties of communication protocols. LySatool is presented in Section 2.5.1, the Open-source Fixed-point Model Checker (OFMC) is presented in Section 2.5.2, and ProVerif is presented in Section 2.5.3. Furthermore, the Beliefs and Knowledge (BAK) tool, which was developed specifically for the GSD framework is introduced in Section 3.

Channel Abstraction, which heavily influenced the GSD framework's security modules, is presented in Section 2.4. Belief Logics play a great role both on the BAK tool and on defining the semantics of the security modules and are, therefore, presented in Section 2.6. SAT and SMT, which are used by the BAK tool, are presented in Section 2.7.

Finally, being able to generate code from the system specification is essential to provide the developer with the code that implements the modeled system. Code generation is introduced in Section 2.8.

1.2 Requirements Analysis

By performing a requirements analysis, the following properties were found to be needed in the framework under development:

- Simple Modeling Language
- The framework must be able to model simple communication systems with security assurances regarding the exchanged messages
- Modeling of the security requirements must be separate from their implementation
- Results from different formal methods tools must be provided
- No specific formal tools knowledge required
- The code skeleton for the communication system must be automatically generated

The GSD Framework, introduced in the Section bellow, was developed having in consideration these requirements.

1.3 The GSD Framework

Part of the GSD framework is strongly inspired by CaPiTo [GNN09] and its main contribution: the successful connection of the abstract specification of communication systems — in CaPiTo's case, Service-Oriented Systems — with the usage of industry standard suites with the verification of the protocol and generation of code. CaPiTo uses LySatool [Buc05] to perform the verification of the system and generates system implementations in the programming language C.

One of the goals of the GSD framework is to apply CaPiTo's main contribution — the separation of concerns regarding the modeling of communication protocols — in a simple and intuitive way. This separation of concerns is an essential part of the GSD framework, which enables the specification, verification, and implementation of communication systems by having different levels of abstraction, step-wise refinement between them and using a simple and intuitive modeling language similar to the Alice and Bob Notation.

The overall structure of the GSD framework is shown in Figure 1.1. The framework is composed by three levels that represent different system perspectives and abstractions:

- The *Abstract Global* level (Chapter 4) is the most abstract level. It is the level where a system designer specifies the communication system. This specification allows the use of security modules, which are presented in detail in Section 4.2, to specify the required security assurances for the data being exchanged in the modeled system;
- A specification of a system in the *Concrete Global* level (Chapter 5) is the result of automatically translating the Abstract Global level specification by unfolding the security modules using plugins. Plugins connect the security modules of the Abstract Global level with the implementations of those modules. For instance, if there is a security module in the Abstract Global level specification that requires some data to be sent in a confidential way, this translation could replace it with an implementation of, for example, TLS. The use of plugins, which as mentioned is inspired by CaPiTo [GNN09], allows for an important separation of concerns when modeling a communication system: the separation between the desired security assurances of the exchanged data and the implementation of those assurances. In Section 5.1, the use of plugins and the flexibility that they provide is discussed;
- A specification at the Concrete Global level is made more concrete by

performing an endpoint projection, resulting in the *Concrete Endpoint* level (Chapter 6). This step separates the specifications into each principal's perspective, resulting in a specification that is closer to the final implementation and to some languages used by the supported verification tools.

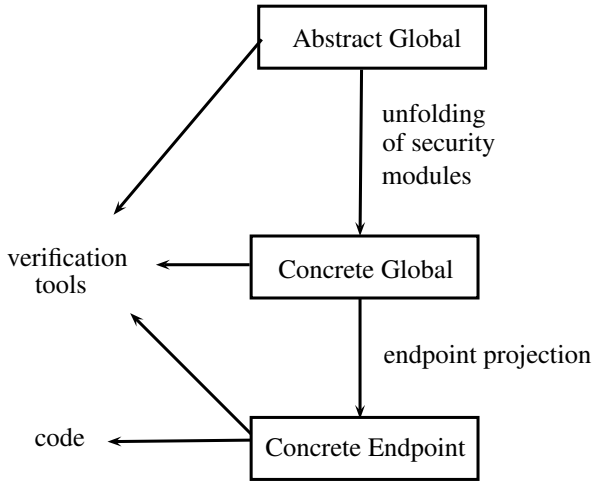


Figure 1.1: Overview of the Guided System Development framework.

Another component of the Abstract Global level is contracts. Contracts represent the desired outcomes of the security modules used in the Abstract Global level. For example, part of the contract that is attached to the confidentiality module is that only the intended recipient of that data is able to see data sent confidentially, i.e., no one else is able to see the plain-text data being exchanged. Contracts allow not only for some preliminary reasoning in the Abstract Global level (discussed in Section 4.3), but also the verification of plugins by verifying the desired outcomes described in the contracts with the outcomes achieved by the different implementations given by the plugins. This is explained in more detail in Section 4.2.3.

The designer is able to choose which implementation of the security modules to use and also perform formal verification of the system after the plugins are automatically replaced by their implementation. Alternatively, the designer has the option of directly modeling the system in the Concrete Global level or Concrete Endpoint level languages, although this will limit the advantages of specifying the system at the Abstract Global level and the amount of formal verification that can be performed. This option could, however, be useful for more experienced system developers or for already implemented systems, in

which case a specification closer to the implementation might be easier to write. In fact, the latter happened in the use case presented in Chapter 8.

Another advantage of this framework is that it allows a protocol designer to specify and test new protocols and provide them as plugins to the system designer. The protocol designer can model a new protocol in the Concrete Global or Concrete Endpoint levels and use the framework to verify its security properties. If found to be secure, the protocol can then be made available to the system designer by defining it as a plugin.

1.3.1 Framework Inputs and Outputs

The modeling language used in GSD is strongly influenced by Alice and Bob notation, a simple and intuitive way of modeling communication systems. The example in Listing 1.1 uses this notation to model a message `msg` being sent by a principal called `Alice` to another principal called `Bob`.

```
Alice → Bob : msg
```

Listing 1.1: Example of sending an (non encrypted) message from `User` to `Board` in Alice and Bob notation.

As previously mentioned, when building a system with secure communications using GSD, the input for the framework is the specification of the system in the Abstract Global level, which includes security assurances for the exchanged messages. Specifying a communication system at this level together with the step-wise refinement will lead to an implementation that provides the specified security assurances.

The outputs of GSD can be divided into two categories: the information from the supported verification tools and the implementation of the modeled system. These are presented in more detail throughout the different Chapters.

1.4 The Message Board

To illustrate the use of the GSD framework, the communication behind a message board is used as an example throughout the thesis to better illustrate the different components of the GSD framework.

The message board allows a user to share a message in an authentic way or not and he might choose to share that message with the entire board or with a specific user, i.e., confidentiality. The combination of these properties gives rise to four different kinds of messages:

1. The message sent can be seen by everybody without any guarantees regarding the identity of the sender,
2. The message sent can be seen by everybody and it has guarantees regarding the identity of the sender,
3. The message sent can only be seen by a particular user without guarantees regarding the identity of the sender, or
4. The message sent can only be seen by a particular user and it has guarantees regarding the identity of the sender.

Throughout the presentation of the GSD framework, a small example of this system will be used where there are three principals (Alice, Bob, and Carsten) who send a message each to the message board (MB). Alice sends a message as described above in `2`, Bob sends a message of type `3`, and finally Carsten sends a message of type `4`.

1.5 Thesis Contribution

The main contribution of my PhD thesis is the Guided System Development (GSD) framework, which was briefly introduced in Section 1.3 and is described in more detail in the remainder of this thesis. The GSD framework has a clear focus on security from the start of the modeling process and it provides automatic translations to languages used by different protocol verifiers using the same system model. These protocols verifiers enable checking the system specification against a variety of security properties and having access to different verifiers, each with its own strengths, allows for a broader range of analyses results. A prototype implementation was developed and is presented in Chapter 7. This prototype provides a proof of concept of the ideas and contributions of this thesis and it was designed with flexibility and extensibility in mind in order to enable the easy implementation of future extensions.

Another important contribution of this work is the Beliefs and Knowledge (BAK) tool, presented in Chapter 3. This tool extends BAN logic [BAN90]

with an explicit Dolev-Yao attacker model [DY83] and uses Z3 [DMB08], a Satisfiability Modulo Theory (SMT) solver, to reason about the security properties of a modeled protocol. Due to the way the attacker is implemented in the BAK tool, it allows for the modeling of different attack scenarios and attacker capabilities in a modular fashion.

1.6 Document Structure

Firstly, an overview of relevant previous work is given in Chapter 2. After that, a presentation of the Belief and Knowledge tool is given in Chapter 3 followed by the presentation of the framework, which consists on the Abstract Global level (Chapter 4, the Concrete Global level (Chapter 5, and the Concrete Endpoint level (Chapter 6). In Chapter 7, details about the prototype implementation of the Guided System Development framework are given, and in Chapter 8 a use case of the framework is presented. Finally, a conclusion is given in Chapter 9 and future work is discussed.

Chapter 2

Previous Work

In this chapter, an overview of related work is given. That includes previous work on frameworks for communication systems (Section 2.1) as well as previous works on tools that provide formally proven specifications and their implementation (Sectio 2.2). Abstraction Languages used to model Service-Oriented Systems are presented in Section 2.3. Channel Abstraction, which heavily influenced the GSD framework's Security Modules, are presented in Section 2.4 and work done on Protocol Verification Tools is reviewed in Section 2.5. After that, Belief Logics (Section 2.6) and SAT/SMT (Section 2.7) are presented. And finally, some techniques for code generation are also discussed, in Section 2.8.

2.1 Secure Communications Frameworks

The GSD framework is not directly targeting Service-Oriented Systems, but the significant amount of work that has been done in the last years in this area makes it important to review. For that reason, different frameworks that target Service-Oriented Systems are presented in these section, together with other frameworks that are targeted more at general communication systems.

2.1.1 CaPiTo

CaPiTo [GNN09] is a framework that aims at connecting the abstract specification of Service-Oriented Systems with the standard protocol suites used in the industry in a independent way, i.e., the description of the system in terms of messages exchanged is separated from the description of which standard suites are going to be used. This is the main novelty of CaPiTo and it helps separating the concerns when modeling a Service-Oriented System. Furthermore, this framework also allows the verification of the modeled protocols and the generation of an implementation of parts of the system.

The CaPiTo framework is composed by three main abstraction levels, which provide the framework with tools to support a specification language, means to verify the security of the specified protocol, and the translation from a protocol specification language into a language that can be executed. The levels are presented in Figure 2.1.

- **Abstract Level** — this level describes the interaction of the services;
- **Plugin Level** — this level adds the identification of the industrial communication protocols to the Abstract Level
- **Concrete Level** — this level adds, through systematic transformation, the cryptographic modeling of the protocols. Furthermore, the necessary annotations are added for compatibility with LySa [BBD+05], which can then be verified with LySatool.

Figure 2.1: The abstraction levels in the CaPiTo framework.

2.1.1.1 Abstract Level

This level focuses on the interaction between the services by abstracting away from the cryptography and also the communication protocols. The language on this level was inspired by CaSPiS [BBDNL08], a process calculus developed in the SENSORIA project [sen10], and its syntax is shown in Figure 2.2. In this language, a service or protocol (P) is built from basic activities such as service invocation ($\overline{n}[\ ].P$) and service provision ($n[\ ].P$), where the latter defines a service that can be invoked by the former. The placeholders ([]) are used in the Plugin level to specify the used security communication protocols. Furthermore, a service can also be built by the restriction of another service ($(\nu\ n)P$), non-deterministic choice between two services ($P_1 + P_2$), parallel composition of

two services ($P_1|P_2$), replication of a service ($!P$), and service return ($\uparrow \langle \vec{e} \rangle.P$). Values ($v, w$) correspond to expressions without free variables such as messages and keys, while expressions (e) can be names (n), variables x, or functions over an array of expressions ($f(\vec{e})$). A definition of a variable is denoted by $?x$ while an application of a variable is denoted by x.

$$
\begin{array}{lll}
v, w & ::= & n \mid f(\vec{v}) \\
e & ::= & x \mid n \mid f(\vec{e}) \\
p & ::= & ?x \mid x \mid n \\
P & ::= & \langle \vec{e} \rangle.P \mid (\vec{p}).P \mid (\nu\ n)P \mid P_1|P_2 \mid !P \mid P_1 + P_2 \mid \\
 & & 0 \mid \overline{n}[\,].P \mid n[\,].P \mid \uparrow \langle \vec{e} \rangle.P
\end{array}
$$

Figure 2.2: Syntax of abstract specifications in CaPiTo

2.1.1.2 Plugin Level

This level is extremely important for the CaPiTo framework because it extends the service interaction from the Abstract Level with information regarding the protocol suites that are going to be used. That is done by adding a list of protocols suites to the specification of the Abstract Level. The syntax of this language is shown in Figure 2.3. As it can be seen, the syntax is an extension of the Abstract Level syntax where the element ps is added, and that element is used to represent the protocol stack. For that reason, the elements are similar to the ones presented in Figure 2.2 with the addition of the protocol stack (ps) in the placeholders ([]) in the service provisioning and invocation constructs. ps represents a list os security communication protocols (pi) that are added to the service communications at this level. Each communication protocol (pi) is defined by its name and its different parameters.

$$
\begin{array}{lll}
v, w & ::= & n \mid f(\vec{v}) \\
e & ::= & x \mid n \mid f(\vec{e}) \\
p & ::= & ?x \mid x \mid n \\
P & ::= & \langle \vec{e} \rangle.P \mid (\vec{p}).P \mid (\nu\ n)P \mid P_1|P_2 \mid !P \mid P_1 + P_2 \mid \\
 & & 0 \mid \overline{n}[ps].P \mid n[ps].P \mid \uparrow \langle \vec{e} \rangle.P \\
ps & ::= & pi \mid pi; ps \\
pi & ::= & name, param_1, \cdots, param_k
\end{array}
$$

Figure 2.3: Syntax of plug-in specifications in CaPiTo

2.1.1.3 Concrete Level

At this level of CaPiTo, the communication protocols and the cryptography are fully modeled. This level is the result of a systematic transformation from the Plugin level and its syntax is shown in Figure 2.4. The transformation from the Plugin level to the Concrete level has two main parts: the removal of the service invocation and provision constructs and the expansion of the protocol stack added in the Plugin level.

In order to remove the service invocation and provision constructs, unique session identifiers are used to distinguish between different services and sessions. The unique session identifiers are added to the service invocation ($\langle r, \vec{e} \rangle.P$), service provision ($(r, \vec{p}).P$), and also to an added service that models communications through an encrypted tunnel ($r : e \blacktriangleright P$).

As for the expansion of the protocol stack, the stack in each service is unfolded using the definitions of the standard protocol suites that are defined in the framework. In order to model these definitions, cryptographic primitives are added to the language. These primitives are asymmetric encryption ($\mathbf{P}_{n^+}(.)$) and decryption ($\underline{\mathbf{P}}_{n^-}(.)$), digital signatures signing ($\mathbf{S}_{n^-}(.)$) and validation ($\underline{\mathbf{S}}_{n^+}(.)$), and also an hash function ($H(.)$).

Finally, the necessary annotations are added in order to provide compatibility with LySa [BBD+05], a process algebra used to describe communication protocols and that can be verified by LySatool [Buc05].

$$
\begin{array}{ll}
v, w ::= & n \mid n^+ \mid n^- \mid \mathbf{P}_{n^+}(\vec{v}) \mid \mathbf{S}_{n^-}(\vec{v}) \mid \mathbf{H}(\vec{v}) \mid f(\vec{v}) \\
e ::= & x \mid n \mid n^+ \mid n^- \mid \mathbf{P}_{n^+}(\vec{e}) \mid \mathbf{S}_{n^-}(\vec{e}) \mid \mathbf{H}(\vec{e}) \mid f(\vec{e}) \\
p ::= & ?x \mid x \mid n \mid n^+ \mid n^- \mid \underline{\mathbf{P}}_{n^-}(\vec{p}) \mid \underline{\mathbf{S}}_{n^+}(\vec{p}) \\
P ::= & \langle r, \vec{e} \rangle.P \mid (r, \vec{p}).P \mid (\nu\ n)P \mid !P \mid r : e \blacktriangleright P \mid (\nu_{\pm}\ n)P \mid \\
& P_1 + P_2 \mid P_1 | P_2 \mid 0
\end{array}
$$

Figure 2.4: Syntax of concrete specifications in CaPiTo

Compared with CaPiTo, the language of the GSD framework is simpler and more intuitive, GSD provides different perspectives of the system, and connects to more verification tools. The GSD framework is targeted at system designers that do not have specific knowledge of formal methods tools. While in CaPiTo the system is only specified in an endpoint view, GSD has both the global view (the more abstract specification of the system) and also the endpoint view of the system, which enables the framework to connect to verification tools with different modeling paradigms.

2.1.2 AVANTSSAR

Another project with similar goals is AVANTSSAR [AVA11], which aims at validating trust and security of Service-Oriented Architectures. AVANTSSAR has ASLAN++ as the specification language that is used to specify trust and security properties of services and their policies and then several automated techniques are used to reason about services, their composition and the described policies.

The AVANTSSAR project is complex and, when compared with the GSD framework, it requires a higher level of expertise to start using. One of the goals of the GSD fraemwork was for it to be a simple and precise tool, targeted at system designers that want to easily model, test, and get the implementation of their system. Another distinction between the two tools is that AVANTSSAR does not provide automatic translation to an implementation language, functionality that is present in the GSD framework and which I believe is important when providing tools to help system designers.

2.1.3 The SPEAR II Framework

SPEAR II [HS99, LVH02] is a security protocol engineering and analysis framework that extends the capabilities of the original SPEAR framework [BDGH97]. It aims at providing a multi-dimensional framework that enables secure and efficient security protocol design and implementation. It has a graphical user interface and it combines formal protocol specification, security and performance analysis, meta-execution and automatic code generation.

One of the main reasons pointed by the authors that makes such a framework needed is that, even in the case where the protocol design is theoretically correct, it does not guarantee the development of a secure crypto-system because the translation from the design phase to the implementation phase is often the most error-prone phase.

The framework comprises of five different, although interconnected, areas:

- **Security protocol design** - provides an interface for the specification of security protocols;
- **Automated security analysis** - analyses the protocol for possible attacks;

- **Performance evaluation and reporting** - provides some information regarding protocol performance;
- **Automatic code generation** - safely generates Java code for the designed protocol;
- **Meta-execution** - makes it possible for the designer to test the designed protocol without the need of a real system.

An overview of the SPEAR II Framework is given in Figure 2.5, where it is possible to see the five different aforementioned areas that constitute the framework.

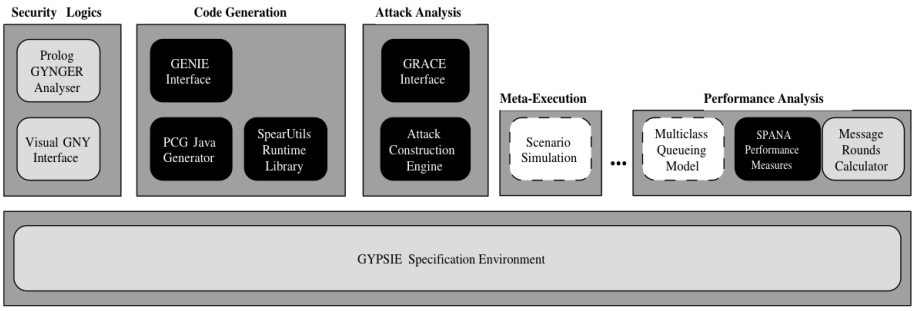


Figure 2.5: The SPEAR II framework

The protocol specification is done through the GYPSIE module and it is used in all the different areas of the framework. The security analysis module has the Attack Construction Engine (ACE) at its core. It is based in a model that converts the reachability problem (finding a state that verifies an error condition) into a constraint solving problem. The model uses the Dolev-Yao attacker model [DY83]. Another tool that belongs to the security analysis module is the Graphical Attack Conduction Environment (GRACE). It assists the user in the ACE analysis by having her specifying, through the GUI, extra protocol properties needed for the analysis. As for the automatic code generation, the Protocol Code Generator (PCG) parses an abstract GYPSIE specification and generates a secure implementation of this specification in Java. The message formatting is specified using ASN.1 [Uni11], making it possible for the implementations to communicate with other non-SPEAR II implementations, as long as they conform to the same standard. PCG does not build an intermediate parse tree from the protocol specification, as done by the majority of the protocol translator tools. It does not need to do that because the GYPSIE module deals with the semantic checking and symbol references. All that is needed then is to directly translate the GYPSIE data structures into Java code. The

code is generated using code templates that contain tokens that are replaced by generated code and it follows the structure shown in Figure 2.6. The communication layer handles all network connections, the Principal Control handles the principal application, Protocol Logic and Message Handling are the protocol implementation, independent of the above layers, and the Security Services layer acts as an interface to the supported cryptographic libraries.

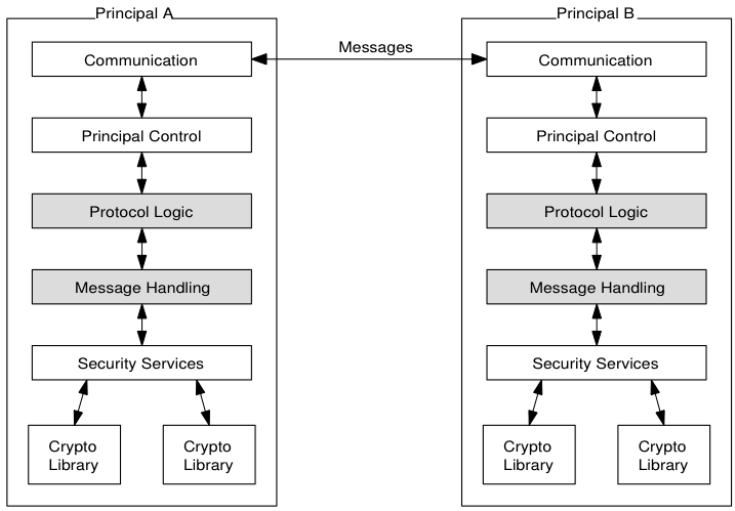


Figure 2.6: Structure of the generated code

The framework does not generate the Java code relying only on the GYPSIE protocol specification, since certain aspects of the protocol, such as checking the freshness of a nonce, are not present in the specification. That is the main reason for the existence of the Generation Environment (GENIE). It provides a way for the protocol designer to control the implementation process by specifying properties such as cryptographic and communication settings and message processing actions. The cryptographic settings allow the user to specify cryptographic libraries and algorithms while the communication settings allow him to specify transport protocol and port numbers.

SPEAR II supports cryptographic protocol specification in a way that "security and performance analysis, meta-execution and automatic code generation [...] can be conducted", providing an important environment for security protocol design.

The GSD framework is simpler than SPEAR II, which entails a smaller learning curve for the system designer, provides the connections to external formal methods tools — which I believe to be important — and has a higher degree

of flexibility. The latter is reflected on how simple it is to extend the GSD framework with new security requirements when modeling system, connections to new formal methods tools, and also output to new languages.

2.1.4 The Automatic Generation, Verification and Implementation of Security Protocols Toolkit

The Automatic Generation, Verification and Implementation of Security Protocols (AGVI) toolkit [SPP01] allows the system designer to describe the security requirements and the system specification. Then a *protocol generator* will create candidate protocols that satisfy the given system requirements. After that, Athena [SBP01], a *protocol screener*, will analyze the protocols and discard the ones that do not satisfy the desired security properties. After each evaluation procedure, Athena returns a counter-example in case the protocol does not satisfy the required properties, or a proof if the properties are satisfied. Finally, a *code generator* automatically translates the formal protocol specification into Java code.

What distinguishes the GSD framework from the AGVI toolkit is the simpler and more intuitive modeling language of the former and also its versatility, i.e., it is simple to extend the GSD framework with new formal verification tools, new output languages, new abstract ways of representing message security properties in the modeling language (extending the security modules presented in Section 4.2), and also new ways of implementing the different security modules.

2.2 Formally Proven Specifications and Their Implementations

There exists more targeted work on the modeling and implementation of secure communication systems that is worth mentioning in this chapter. When compared to the GSD framework, these tools support a more concrete modeling language and target a specific verification tool.

2.2.1 FS2PV

FS2PV [BFGT08] is a tool that derives a formal model from a protocol code and symbolic libraries. The protocol code is written in a first-order subset of

F#, which the authors named F. It is a subset with simple formal semantics facilitating model extraction, with primitives for communication and concurrency, but it does not allow higher-order functions and some imperative features, such as recursion.

The translation is made to applied π-calculus, which can then be verified by ProVerif (Section 2.5.3). When the theorems are proved with ProVerif, it means that those properties are present in the source code programs defined in F and, therefore, that the communication protocol is secure.

2.2.2 Other

Bhargavan et al. [BFGP04] defined TulaFale, a language to describe security protocols based on SOAP. TulaFale uses *pi*-calculus as its core for describing the different principals and extends it with XML syntax, logical predicates, and correspondence assertions. The latter are used to specify the authentication goals of the system. A description of a communication system in TulaFale is translated into applied *pi*-calculus in order to verify the authentication properties of SOAP protocols with ProVerif.

Swamy *et al.* [SCF+11] developed a dependently typed language (F*) aimed at secure distributed programming. Programs written in F* are translated to .NET byte-code.

Other more recent work [CB12] enables the verification of a protocol with CryptoVerif [Bla08] and then translates that specification into OCaml.

Another recent work by Modesti [Mod11] presents AnBx. An extension of AnB, which presented in Section 2.3.2, that targets the declarative modeling of distributed protocols, also uses OFMC to verify that specification and further translates that to Java.

2.3 Abstraction Language for Service-Oriented Systems

Another important part of the research presented in this thesis concerns the choice of language used to model the communication systems. For that reason, a review of the state-of-the-art in this area is extremely important.

2.3.1 KLAIM

KLAIM [DNFP98] is a process calculus that supports a programming paradigm where processes move between different computing environments. This is different from the usual way of programming where data is what is explicitly moved between the different environments, i.e., where data is sent and received between different entities in a network. In KLAIM, however, the communication over the network is not modeled by explicitly sending and receiving data, but by having the processes reading and writing in the different tuple spaces.

With KLAIM, the network structure is explicitly modeled and the language provides coordination mechanisms to control the interaction between processes. KLAIM has Linda [Gel85] at its core and extends it with multiple tuple spaces and processes' operators. Linda is a coordination language that uses an asynchronous and associative communication mechanism based on a tuple space, a shared global environment that contains sequences of information items.

As previously mentioned, the use of a tuple space is extended in KLAIM: instead of having a single tuple space, KLAIM uses multiple tuple spaces, one for each locality. The other extension from Linda is the definition of operators for process construction and composition, inspired by Milner's CCS. Furthermore, a Type System is used to describe the intentions of processes regarding other localities, such as read, write, and execute.

2.3.2 AnB

AnB [MÖ09] is a language based on the Alice and Bob notation (Section 1.3.1) and it is defined over an arbitrary algebraic theory. Based on the algebraic properties of its operators, it defines unambiguously how a specified protocol is supposed to be run by honest agents. The specification of a protocol in AnB consists of four sections, presented in Figure 2.7.

The semantics of AnB are expressed in a formalism called IF and more information on this translation and the use of AnB is given when presenting the Open-source Fixed-Point Model Checker in Section 2.5.2.

- **Types** - all the identifiers used in the protocol are declared. If an identifier is a variable, it must start with upper-case, if it is a global constant or function, it must start with lower-case. Furthermore, the agents present in the protocol are also named and are called roles. Whether the name starts with upper-case, then the agent is considering dishonest in the protocol analysis, otherwise it is consider honest;
- **Knowledge** - the initial knowledge for each role is specified here. If a variable does not appear here, it is assumed to be freshly created;
- **Actions** - in this section, the list of exchanged messages is specified, describing the normal execution of the protocol;
- **Goals** - the goals that the protocol wants to achieve are described in this last part.

Figure 2.7: Sections of the AnB code

2.3.3 CaSPiS

CaSPiS [BBDNL08] was developed with the goal of making the modeling of Service-Oriented Systems easier. This was motivated by several important aspects of Service-Oriented Systems such as service autonomy, client-server interaction, and the orchestration of services.

CaSPiS is a process algebra where, for the reasons mentioned above, sessions and pipelines have a crucial role. Sessions are two-sided and allow protocols to be executed by each side and pipelines enable the orchestration of data flow produced by the different sessions.

2.4 Channel Abstractions

Channel abstractions strongly influenced the security modules present in the GSD framework (Section 4.2) and, therefore, they are presented here.

Channel abstractions were first introduced by Maurer and Schmid [MS96]. This work establishes secure channels at a high level of abstraction: channels are regarded as an abstraction of the transport of a message from an input to an output and the different possibilities regarding the exclusiveness of access to the input and output give rise to different kinds of channels:

- **Authentic**: exclusive access to the input;
- **Confidential**: exclusive access to the output;
- **Secure**: exclusive access to both the input and the output;

In the literature, a common motivation for the use of abstract channels is the necessity of separating concerns when modeling and developing systems. Channel abstractions allow one to have a simpler vision of the system (focused on its functionality) without the need to consider the security details at such an early stage of development.

In the aforementioned work [MS96], Maurer and Schmid characterize a channel by its direction, its time of availability and its security properties. Furthermore, they identify the security goals that might be relevant in a distributed systems, such as secrecy, authentication and non-repudiation.

Establishing security in a distributed system, as the authors describe, consists of two phases:

- *Initialization Phase*: where the set up of security parameters occurs in a secure environment;
- *Connection Phase*: occurs over insecure channels and cryptographic techniques are applied in order to achieve the required security level in the channels.

Therefore, a protocol transforms the set of security parameters and the insecure channels into a set of secure channels, specified by the security requirements. A communication channel can be seen, at a high level of abstraction, as means for transporting a message from a source entity to a destination entity.

Lastly, the authors state that the paper was not intended to be applied for the design or security verification. Its original main goal was to illustrate interesting aspects of distributed systems security.

Another important work on channel abstractions is by Abadi et al. [AFG02] where three main contributions are given: a simple, high-level language (sjoin-calculus), which is a variant of the join-calculus [FG00] with constructs that enable the creation and use of channels; a translation from that high-level language to a low-level language that includes cryptographic primitives. This translation maps communication on secure channels to encrypted communication on public

channels; and finally, a correctness theorem for the translation that allows the reasoning about programs with the high-level language.

Their main motivation for this work is the fact that application code, in their view, should not be concerned with lower-level details. To avoid that, they defend using abstractions and services that encapsulate cryptographic operations. In other words, the high-level abstraction hides the difficulties and details of network security: systems run as if they were on the same machine and security is provided for them. This is also one of the main reasons for the use of the security modules on the GSD framework (they are introduced in Section 4.2).

In the translation from the sjoin-calculus to join-calculus, each channel X is mapped to a key pair $\{X^+,X^-\}$. The authors present, in fact, two kinds of translations: a compositional translation from join to sjoin and another one that puts the resulting join processes in a context that filters every communication with the environment. In this translation all communications are equally protected, as if all the communications had a secure channel as described above.

Adao and Fournet [AF06] present a variant of the π-calculus that adds secure communication, mobile names and high-level certificates but does not have explicit cryptography. The security properties of a system are studied in the presence of an arbitrary, abstract adversary, recurring to trace properties and observational equivalences. The verified security properties can be achieved, the authors argue, with some care in a concrete setting with standard cryptographic primitives. They aim at building security abstractions with formal semantics and sound computational implementations by developing a sound and complete implementation of a distributed process calculus. Their system provides uniform protection of all messages. They denominate that kind of protection as authentication, but it resembles, in fact, the secure channels previously introduced by Maurer and Schmid.

Bugliesi and Giunti [BG07] present a secure implementation of a typed π-calculus where capability types are employed to realize the policies for the access control to communication channels. Therefore, the different channel types (authentic, confidential, and secure) are implemented by means of resource control of the communication channel.

A translation is given from high-level type capabilities to an implementation that has term capabilities protected by encryption keys that are only known by the intended principals. Typed equivalences of the high-level calculus correspond to the untyped equivalences of the low-level cryptographic calculus.

The channel types, `ch(.)`, are used to model the different security properties and represent the access to the channel. The types are a combination of reading(`r`)

and writing(w) capabilities and a lattice is defined over those values, as it is show in Figure 2.8.

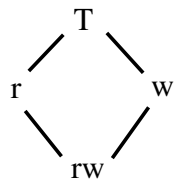


Figure 2.8: Channel types access capabilities lattice

Bugliesi and Focardi [BF10] aim at identifying abstractions that are adequate for, on one hand, high-level programming and specification and, on the other hand, security analysis and verification. That was done because, in their opinion (and as we have already seen, in the opinion of many others), there are two important security areas in distributed systems where attention should be payed to: having formal low-level specifications and allowing those low-level details to be abstracted away when programming those systems, allowing the focus to be on the systems functionality. The calculus used is one previously presented by the same authors [BF08] and that calculus is analyzed with various bi-simulation based security equivalences, with different attacker capabilities, and also with the attacker having an increased control over the network. The same three types of channels commonly used by other authors are used: authentic, secret (sometimes called confidential) and secure. Furthermore, a new set of security abstractions is provided and the primitives of these security abstractions can be seen as a Kernel API.

Mödersheim and Viganò [MV09b] assume three kinds of channels: authentic, confidential and secure (the commonly used combinations introduced by Maurer and Schmid[MS96]) that can be used either as assumptions or as goals. Whenever a channel is used as an assumption, the communication being modeled relies on that channel and the security properties it represents. On the other hand, if a channel is used as a goal, it means that the modeled communication establishes the security properties of that channel. The channels (shown in Figure 2.9) are modeled in the AnBbullet language, which extends AnB[MÖ9] with support for channel specification. It is worth noting that the use of the bullets in this language can be interpreted as modeling the exclusiveness of access to the communication channel mentioned by Maurer and Schmid [MS96].

The authors present two models for channels as assumptions. The ideal channel model (ICM), which represents the ideal functionality of the channel, and the cryptographic channel model (CCM), which represents the implementation of the channels by cryptography. As the authors claim, relating these two models

- A $\bullet\rightarrow$ B : M - authentic channel from A to B;
- A $\rightarrow\bullet$ B : M - confidential channel from A to B;
- A $\bullet\rightarrow\bullet$ B : M - secure channel from A to B (channel is both authentic and confidential).

Figure 2.9: AnB channels

enables one to get insight to the meaning of channels as assumptions and it also allows for interchangeability in analysis tools.

Furthermore, the authors also propose a new, stronger, authentication goal because the standard one is too weak. They also identify a general composition problem, the interpretation of messages in different protocols that are similar and that might enable messages to be read in places where they were not supposed to. These two issues are discussed in more detail in Section 4.2.1. In fact, that stronger authentication goal (with freshness guaranteed) was taken in consideration when AnBx [BM11] was developed, which directly supports the modeling of authentic and secure channels with freshness guarantees and further translates those into the AnB specification used by OFMC.

2.5 Protocol Verification Tools and Languages

In this section, the state-of-the-art of tools and languages targeted at protocol verification is presented. Firstly, the external tools used in the GSD framework are reviewed, LySatool in Section 2.5.1 and OFMC in Section 2.5.2. Then ProVerif is also reviewed in Section 2.5.3.

2.5.1 LySatool

LySatool [Buc05] is a tool that performs security analysis of protocols described in LySa (presented below). LySatool is implemented in Standard ML and it starts by encoding the flow logic specification (written in LySa) into an ALFP formula, where the sets of terms are encoded by tree grammars. ALFP is an expressive fragment of first order predicate logic that is interpreted over a finite universe. Succinct Solver [NNS+04] uses the ALFP specification to compute the least solution to flow constraints. LySatool receives the LySa specification

of a system as input and performs a static analysis in the presence of a Dolev-Yao [DY83] attacker. If the output of the analysis is positive, then the protocol satisfies the specified notations. On the other hand, if the output is not positive, then an attack might exist. In that case, further analysis has to be performed in order to find whether it was a false negative or if a real attack was discovered. The grammar for the processes that LySatool accepts as input is described in 2.10.

```
proc        ::= ( proc )  |  < term* > . proc  |  ( term* ; var* ) . proc  |
                decrypt term as { term* ; var* } : term orig in proc  |
                decrypt term as {| term* ; var* |} : term orig in proc  |
                ( new name ) proc  |  ( new +- name ) proc  |
                ! proc  |  proc | proc  |  0  |
                let identifier subset iset in proc  |  |_{ assign } proc  |
                ( new_{ assign+ } name ) proc  |  ( new_{ assign+ } +- name ) proc

term        ::= ( term )  |  { term* } : term dest  |  {| term* |} : term dest  |
                name  |  namep  |  namem  |  var

name        ::= identifier subscript

namep       ::= identifier + subscript

namem       ::= identifier - subscript

var         ::= identifier subscript

subscript   ::= _{ index* }  |  ε

index       ::= identifier  |  number

iset        ::= { index* }  |  iset union iset  |
                NATURAL1  |  NATURAL2  |  NATURAL3  |
                NATURAL01  |  NATURAL02  |  NATURAL03  |  ZERO

assign      ::= index in number

dest        ::= [ at cryptopoint dest { cryptopoint* } ]  |  [ at cryptopoint ]  |  ε

orig        ::= [ at cryptopoint orig { cryptopoint* } ]  |  [ at cryptopoint ]  |  ε

cryptopoint ::= identifier subscript  |  CPDY
```

Figure 2.10: LySatool grammar

2.5.1.1 The Language - LySa

LySa [BBD+05] is a process algebra based on standard protocol narrations extended with annotations. These annotations were added in order to improve the standard protocol narrations, which are usually imprecise about (important) details related to the deployment of the described protocol. With this addition, the protocol narration becomes clearer and with less space for ambiguity when the analysis is performed.

LySa is based on the Spi-calculus [AG99], which extends the π-calculus [MPW92a, MPW92b] with the use of encrypted values. LySa is similar to these calculi, although it differs from them mainly in two aspects:

- LySa's underlying model does not use individual channel communication between the different principals, but one main channel where all the communications go through. This results in a model closer to what we want to represent and analyze, since an attacker might have the capability of eavesdropping the communications;
- In LySa, the tests associated with input and decryption are performed using pattern matching.

In the LySa specification only the legitimate part of the system are described. That specification shall be explicit about using the protocol in a general setting, where many principals might use the protocol simultaneously. Each of the principals, denominated $I_1, ..., I_n$ may serve as an initiator (usually A), as a responder (usually B) or even as both (which means that, when modeled this way, the principal runs the protocol with himself). Furthermore, LySa adopts the assumption of perfect cryptography, which can be approximated in a real implementation by using a secure cryptographic system.

The attacker outside the legitimate part is given the name I_0. To model this attacker, a formula inspired in the studies of Dolev and Yao[DY83] was defined. The attacker has some initial knowledge, it can expand that knowledge by eavesdropping the network, by decrypting messages with keys it already knows, or by encrypting new messages using the keys he knows, and it may engage in new communications.

2.5.1.2 LySa Syntax

The LySa language is formed by terms and processes. Whenever terms without free variables are present, we call them values. It has structures for representing input and output on the network, parallel composition of processes (the principals of a protocol are modeled running in parallel), restriction of values to specific processes, replication and symmetric and asymmetric cryptographic operations (encryption and decryption). Furthermore, the end of a process is represented by 0. The full syntax is shown in Figure 2.11.

E	::=	*terms*	
	n	name	
	m^+, m^-	public and private keys	
	x	variable	
	$\{E_1, ..., E_k\}_{E_0}$ $[at\ l_1\ dest\ l_2]$	symmetric encryption ($k \geq 0$)	
	$\{\|E_1, ..., E_k\|\}_{E_0}$ $[at\ l_1\ dest\ l_2]$	asymmetric encryption ($k \geq 0$)	
P	::=		*processes*
	0		nil
	$\langle E_1, ..., E_k \rangle.P$		output
	$(E_1, ...E_j; x_{j+1}, ..., x_k).P$		input with matching
	$P_1 \| P_2$		parallel composition
	$(\nu\ n)P$		restriction
	$(\nu_{\pm} m)P$		restriction(key pair)
	$!P$		replication
	decrypt E **as** $\{E_1, ..., E_j; x_{j+1}, ..., x_k\}_{E_0}$ $[at\ l_1\ orig\ l_2]$ **in** P		symmetric decryption with matching
	decrypt E **as** $\{\|E_1, ..., E_j; x_{j+1}, ..., x_k\|\}_{E_0}$ $[at\ l_1\ orig\ l_2]$ **in** P		asymmetric decryption with matching

Figure 2.11: LySa syntax description

One can note that when using pattern matching — in decryption and input — a semi-colon separates the components that are being matched (on the left) from the variables to which the received values are assigned to (on the right). Also, it is possible to see the presence of the assertions for origin and destination, which decorate the protocol text with cryptopoints, important for the analysis. In these assertions, l_1 represents a cryptopoint and l_2 represents a list of cryptopoints in the protocol. In LySa, cryptopoints are used as a way to specify where a term is being encrypted and where it is supposed to be decrypted. This description makes it easier for the analysis to detect some of the security violations.

2.5.2 The Open-source Fixed-point Model Checker

The Open-Source Fixed-Point Model-Checker [MV09a], also known as OFMC, is a symbolic security protocol analyzer that detects attacks on the protocol

and both performs a bounded session verification by exploring the transition system of the protocol representation and an unbounded session verification using fixed-point based techniques.

OFMC's primary input language is the Intermediate Format (IF) [AVI03] specification, which describes a security protocol as an infinite-state transition system using set rewriting. The tool also accepts AnB [MÖ9] as input, which is then automatically translated to IF, defining a formal semantics for AnB in terms of IF. This translation is performed by defining a state machine for each principal described in the protocol.

OFMC uses several techniques that significantly reduce the search space of a protocol without introducing, or excluding, any attacks. Two of the major used techniques are *lazy intruder* and *constraint differentiation*. The first is a symbolic representation of the intruder while the latter is a general search-reduction technique.

2.5.3 ProVerif

ProVerif [Bla09] is a protocol verification tool that provides a fully automatic way of verifying, among others, authentication and secrecy of communication protocols. It does so by verifying correspondences in the modeled protocols with an unbounded number of sessions. Correspondences are properties that represent associations between certain events, enabling the reasoning of authentication, secrecy, and other properties. Correspondences can be used to describe, for example, that if one event happens, then another event has to have happened at some point in the past.

Blanchet's definition of authentication is that, when a protocol is used to authenticate Alice to Bob, then when Bob thinks he is talking to Alice, he really is talking to Alice. The use of correspondences for verifying authentication is in fact similar to the notion of agreements (injective and non-injective) used by Lowe [Low97]: an event is registered when Alice starts the communication and the correspondent final event is registered by Bob when finishing the communication. Lowe's non-injective agreement corresponds to a protocol where two or more end events might occur for each start event. On the other hand, Lowe's injective agreement is present when there is a one-to-one correspondence between the start and end event.

Blanchet defines the secrecy of a value M by the impossibility of an attacker to obtain that value M after the protocol is executed. In correspondences, that is modeled by an event that specifies that the attacker is able to see M and secrecy

is achieved when that event does not occur after executing the communication protocol.

The protocol is modeled in the applied π-calculus, a process calculus presented by Blanchet [Bla02] that extends π-calculus with cryptographic primitives. The last version of ProVerif [Bla09] further extends the applied π-calculus with events, which model the correspondences. The specification of a communication protocol in applied π-calculus with the events annotations is translated into a set of Horn clauses that is used by a resolution-based solver to analyze the modeled protocol and prove the different properties modeled by the correspondences. If a proof is not found, ProVerif tries to reconstruct an attack using a derivation of the Horn clauses.

The verification performed in ProVerif might not terminate since the tool performs protocol verification for an unbounded number of sessions, which was shown to be an undecidable problem by Durgin et al. [DLMS04]. This means that not all the protocols can be verified. The way ProVerif and other verifiers minimize this is by using techniques, such as abstract interpretation, that make the system easier to reason about. This leads to a verification process that is not complete (i.e., false attacks might be found), but sound (when an attack is not found, the specified properties are verified). The abstractions performed by ProVerif simplifies the problem of verifying communication protocols. However, the simplifications used by ProVerif do not produce a finite state space (due to the unbounded number of sessions) and thus, despite these approximations, the tool may still not terminate. In practice, however, ProVerif terminates (and quickly) on many real-world protocols. On the other hand, approximations are the source for non-completeness: the price to pay for termination very often is that sometimes false positives are detected.

Weidenbach [Wei99] can be cited as the main inspiration for Blanchet's work, since he proposed the application of resolution-based theorem proving to the problem of protocol verification.

Selinger [Sel03] showed the duality of theorem proving and satisfiability for the problem of verifying protocols: a single model for a protocol and a security property is needed to establish that the protocol satisfies the property. Selinger also uses one single predicate to formalize the knowledge of the attacker.

2.6 Belief Logics

In 1990, Burrows, Abadi, and Needham [BAN90] introduced the BAN logic for the reasoning of beliefs in distributed systems. This logic describes the beliefs of the honest principals of the system and that is done by annotating an idealized protocol with expressions that represent the beliefs of the principals. The reasoning of the protocol properties is done by checking which formulas (derived from the initial assumptions of the protocol and its annotations) hold as conclusions.

The idealization of the modeled protocol is a usual step when one wants to use the BAN logic. This step is not systematic and it can lead to, for example, having the same idealized protocol of variations of a protocol where those variations are crucial to its security. This step is needed when using the BAN logic in cases where it is not clear what the role of the exchanged elements is in the communication protocol. This step is not problematic in the GSD framework because the system is modeled at a higher level of abstraction and the elements in the more concrete levels are previously defined. For this reason, the roles of the exchanged elements are also predefined and allow for the analysis of the system.

In this Section, the BAN logic is introduced together with other work that reviews the logic.

2.6.1 A Logic of Authentication

The BAN logic describes the beliefs of the trustworthy parties involved in the described protocol, which allows for the reasoning about knowledge in distributed computation and, more specifically, in authentication protocols.

The BAN logic was created with the aim of answering several questions with the help of formal methods. These questions are, for example:

- Does the given communication protocol work?
- What does it achieve?
- Does one protocol need more initial assumptions than the other?
- Does a protocol do anything unnecessary?

The logic operates at an abstract level and, therefore, does not consider errors in the implementation of the protocol. It introduces notation that allows for the expression of different kinds of belief. The authors introduce predicates like `P believes in X`, `P sees X`, `P said X`, `P controls P`, among others that say, for example, that an element is fresh and express other beliefs regarding the usage and secrecy of keys. These predicates are presented in more detail in Section 3.1.1.

This logic uses idealized protocols, which provide a more explicit and cleaner information about the usage of the elements of the messages. The translation to the idealized version of a protocol tries to remove any ambiguities that might exist regarding the usage of the different elements of the exchange messages, making it easier to derive a practical encoding from these idealized protocol. As previously mentioned, one of the main problems of BAN logic is the fact that the idealization process is not a systematic process. Not having a systematic way of generating the idealized protocol makes the analysis of the protocol error-prone. This happens due to possible ambiguities and different interpretations of non-idealized protocols that might be translated in different ways into the idealized version. That is clearly seen when two different versions of a protocol, with small but crucial differences, originate the same idealized protocol.

The analysis of the protocol modeled with the BAN logic relies heavily on its annotations. An annotation of a protocol is a sequence of comments about the beliefs of principals and what they see. For the analysis to be successful, initial assumptions must be made regarding the keys that are initially shared, which values are fresh, who are the trusted principals, etc. Then, a protocol is verified by analyzing the initial assumptions together with the annotations in order to prove that a given formula holds as conclusion.

2.6.2 Reviews of the BAN logic

Syverson [Syv91] defends that there is some confusion regarding the goals and capabilities of the BAN Logic. As he notes, the BAN logic was designed for reasoning about the evolution of the belief and trust of the participants in a protocol. He goes on to explain that one uses formal semantics to help gain understanding of the abilities and limitations of logics and that one should be careful not to analyze the wrong properties while using a specific logic. Syverson highlights that the authors of the BAN logic say that using just their notion of belief would be harmful in the study of security properties and he defends that the BAN Logic was extremely important in establishing requirements of any logic for protocol analysis, but should not be used as a formal method for that purpose.

Liebl [Lie93] explains that there are two kinds of logics for protocol analysis: the Logic of Knowledge and the Logic of Belief. While the former concerns the view of the intruder and is related to the security of a protocol, the latter concerns the views of the legitimate principals and is related with the functionality of the protocol. BAN Logic is Logic of Belief and the author describes it as a logic where protocols are viewed at a high level of abstraction, that cannot discover flaws that might occur when users are able to play multiple roles. Furthermore, Liebl also notes that the BAN logic's lack of systematic translation from the protocol specification to the idealized protocol is a problem.

Boyd and Mao [BM93] state that it is easy for the BAN Logic to approve protocols that are in practice unsound. They argue that the main reason for that is the lack of precision of the idealization process, its inability to express certain events. The main point for this argument is the fact that variations of protocols may not be easily distinguishable in BAN logic, but they are critical for the protocols security. The authors expand this idea by proposing that a logic that does not require idealization should be developed. They do not, however, put forth such a logic.

2.7 SAT and SMT

SMT is an important part of the Beliefs and Knowledge tool that is presented in Section 5.3.1. In this section, SAT is firstly introduced and followed by presentation of SMT, which is an extension of SAT.

There are several situations where one needs to verify the fulfillment of several different constraints. For example, when determining the lecture schedule that has to have in consideration the availability of rooms, teachers, and students. One way of solving that is by using propositional logic formulas to express those constraints on variables. In their simplest form, these variables are Boolean values (either `true` or `false`) and this gives rise to the Boolean Satisfiability problem, also know as Propositional Satisfiability or SAT: for a given propositional formula over Boolean variables, find a satisfying assignment for the variables or give a proof that no satisfying assignment exists [MZ09].

The propositional formulas are constructed with three operators: `AND`, `OR`, and `NOT`. Having two formulas, `f` and `g`, the semantics of the operators is shown in Figure 2.12.

- NOT f: evaluates to true if f evaluates to false and evaluates to false if f evaluates to true;
- f AND g: evaluates to true when both f and g evaluate to true, and evaluates to false otherwise;
- f OR g: evaluates to FALSE when both f and g evaluate to false, and evaluates to true otherwise.

Figure 2.12: SAT operators semantics

An example of a SAT formula is:

$$(x_1\ OR\ x_2)\ AND\ (NOT\ x_3)$$

If the variables are assigned with the values $x_1 = true$, $x_2 = false$, and $x_3 = false$, then the formula evaluates to true, i.e., a satisfying assignment for the formula was found. If $x_3 = true$, then the formula evaluates to false, regardless of the values of x_1 and x_2. It might be the case that no satisfying assignment of variables exists for a formula. Consider, for example, the simple formula:

$$x_1\ AND\ (NOT\ x_1)$$

There is no assignment for the variable x_1 that satisfies this formula, i.e., that makes the formula evaluate to true. In this case, the problem is over-constrained and the formula is said to be unsatisfiable. It is worth nothing that validity is dual to unsatisfiability and that, therefore, knowing that $x_1\ AND\ (NOT\ x_1)$ is unsatisfiable also allows us to reason about its negation:

$$NOT(x_1\ AND\ (NOT\ x_1))$$

This formula is in fact valid, i.e., satisfiable in every model because, as it was shown above, its negation is unsatisfiable.

Determining whether there is a satisfiable assignment of the variables in a formula is an NP-complete problem, but practical advances have enabled SAT solving to be efficient despite its formal complexity. One of those advances, widely used by SAT solvers, is the systematic search approach. The search space used in this approach is a binary tree where each vertex represents a variable and

each of its edges represents the possible values for that variable (either `true` or `false`). Therefore, each of the paths from the root to a leaf corresponds to an assignment of values to each of the variables in the formula. The most well-known algorithm using this systematic search approach is DPLL [DP60, DLL62].

Satisfiability Module Theory (SMT) [DMB11] extends SAT for cases where Boolean values are not enough. This extensions is made by having the variables governed by one or more underlying theories, such as Linear Integer Arithmetic, Difference Logic, Arrays, Lists, and Uninterpreted Functions.

In SMT, the atoms in one theory (or in a combination of different theories) are abstracted into Boolean values and then the SAT engine searches for an assignment that satisfies the formula. If no assignment is found, then the SMT formula is unsatisfiable. On the other hand, if an assignment is found, theory solvers are used to check if the formulas that were abstracted are valid under the different theories in use.

Considering the following SMT formula:

$$NOT(x < 5)\ AND\ (x < 3\ OR\ x = 10)$$

The atoms of the formula are then abstracted into Booleans, which results in the formula:

$$NOT(v_1)\ AND\ (v_2\ OR\ v_3)$$

A model for this formula is, for example, $(v_1 = false, v_2 = true, v_3 = false)$. After having the model, the theory solver is used to check if the atoms hold in relation to those values in the underlying theories, and, as one can see, this is not the case due to the values of v_1 and v_2. On the other hand, the model $(v_1 = false, v_2 = false, v_3 = true)$ holds in the underlying theory.

2.8 Code Generation

Being able to generate code from the system specification is essential to provide the developer with the code that implements the modeled system. Firstly, a commonly used tool for language translation is presented in Section 2.8.1 on the following page. After that, a solution for language translation using a functional language is presented in Section 2.8.2 on the next page.

2.8.1 ANTLR

ANTLR, which stands for ANother Tool for Language Recognition, is a parser generator developed by Terence Parr [Par07]. It is, in fact, a framework for generating recognizers, interpreters, compilers and translators based on grammatical descriptions, and besides providing support for building lexers and parsers, it also supports tree construction and tree walking. Parsers can generate abstract syntax trees (AST) which can be further analyzed with tree walkers. The framework also has a tight integration with the StringTemplate [Par09], which makes it suitable for translating from one language to another.

One flexible option when using ANTLR for code to code translation is to make it generate the lexer, the parser, and the tree walker that performs the translation. A possible process of using ANTLR as a language translator is shown in Figure 2.13.

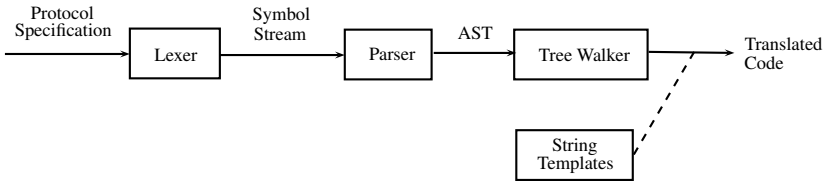


Figure 2.13: ANTLR possible translation process

2.8.2 Code generation in F#

F# is a functional programming language developed by Microsoft Research. It is a scripted, imperative object oriented language available in the .NET platform that combines type safety, succinctness, performance expressivity and scripting.

When using F#, one possibility is to use FsLex and FsYacc to generate the lexer and the parser and those will output an abstract syntax tree. Having that syntax tree and F# being a functional language, it makes it fairly simple to analyze and translate that abstract syntax tree. More details about that process are given in Section 7.2.

Chapter 3

The Beliefs And Knowledge Tool

The Beliefs and Knowledge (BAK) tool was developed to verify the security of communication protocols by reasoning about the beliefs and the knowledge that the different principals involved in a communication system acquire throughout the message exchange. The tool uses the Z3 SMT Solver [DMB08] and adds an extra layer that facilitates the modeling of message exchanges and the reasoning about those messages. This extra layer is composed of a set of predefined system and inference rules, presented in Section 3.1.1.

3.1 The Logic

The logic used in this tool is strongly inspired by belief logics such as the BAN logic [BAN90]. This logic focuses on the knowledge the legitimate principals are able to infer from a message exchange, and is targeted at reasoning about authentication. It is not, however, that easy to directly reason about confidentiality with such logics. This is the case because confidentiality concerns what some attacker might, or might not, be able to see from a message exchange and the BAN logic was not design to explicitly reason about attacker knowledge.

There are several approaches that one can take to reason about confidentiality in this case. In the work presented here, the normal belief logic is extended with explicit reasoning about the attacker knowledge: the beliefs he is able to infer from the message exchange and also what he is able to see, and what he is *not* able to see regarding the exchanged messages, which is of great importance when reasoning about confidentiality. This approach was chosen mainly because it results in a model that is simple and easy to reason about.

Furthermore, in this work, attackers are considered to have the capabilities of the Dolev-Yao attacker [DY83], i.e., he is not only able to see all the exchanged messages but also capable of initiating protocol communications with legitimate principals and to generate new messages based on acquired knowledge.

3.1.1 Logic Predicates

The logic is composed by principals and elements (all the terms that can be sent from one principal to another). The different predicates in the logic are shown in Figure 3.1. X and Y are used to represent principals and el, $el1$, and $el2$ are used to represent elements.

- **Sees(X,el)** - X sees a specific element el;
- **Believes(X,el)** -X believes in a specific element el;
- **Said(X,el)** - X said, at some point in time, a specific element el;
- **Says(X,el)** - X *recently* said element el;
- **Fresh(el)** - expresses the fact that the element el is fresh;
- **MsgSent(X,Y,el)** - X sent element el to Y;
- **Conc(el1,el2)** - element that is the concatenation of two elements el1 and el2;

Figure 3.1: Predicates in the logic used in the BAK tool

The first five predicates in Figure 3.1 are taken from BAN logic while the last two predicates were defined specifically for the BAK tool, since they make it simpler to reason about the system. The semantics of the predicates are given by the rules presented below.

3.1.2 The Logic Rules

There are several rules that constitute the core engine of the BAK tool. These are the system and belief rules, the cryptography rules, and the attacker model rules. They specify the way the principals are able to exchange, construct, and read the messages, and how they acquire the different beliefs and they are shown in the sections below, where X and Y represent specific principals, *attacker* represents the attacker, el represents an element, m represents a message in plain-text, $Enc(el, k)$ represents the encryption of the element el with the key k, and $pubKey(X)$ and $privKey(X)$ represent, respectively, the public-keys and private-keys of a principal X. Details on the implementation of these rules are given in Section 7.3.2.

3.1.2.1 System and Belief Rules

- all the principals are distinct;
- all principals see the public keys of the other principals. This is an abstraction of an infrastructure that allows the principals to retrieve the public key of other principals;
- rules used to implement the required domain restriction. More details are given in Section 7.3.3.
- **MsgSent(X,Y,el)** $\Longrightarrow$ **Sees(Y,el)** $\wedge$ **Sees(attacker,el)** - when a message is sent between two principals, both the intended receiver and the attacker are able to see the message. This rule exists to enable the reasoning of beliefs in the system. If reliability of the network was not assumed (for example, if the attacker had control over the network), it would not allow for the reasoning of the system security properties. This reasoning is based on the acquired beliefs and on which elements are seen by the different principals and, since the attacker could retain all the sent messages, it could lead to a situation where no element would be seen by any of the principals, affecting the reasoning of the security properties;
- **Believes(Y,Said(X,el))** $\wedge$ **Believes(Y,Fresh(el))** $\Longrightarrow$ **Believes(Y,Says(X,el))** - If Y believes that X sent an element at some point in time and if he believes that that element is fresh, then he is entitled to believe that X recently sent that element;
- **Sees(X,Conc(el1,el2))** $\Longrightarrow$ **(Sees(X,el1)** $\wedge$ **Sees(X,el2))** - if X sees the concatenation of two elements, he also sees the two elements;

- **(Believes(X,Fresh(el1)) $\vee$ Believes(X,Fresh(el2)) $\wedge$ Sees(X,Conc(el1,el2)) $\Longrightarrow$ Believes(X,Fresh(Conc(el1,el2)))** - if X believes that one of the elements in a concatenation is fresh, then he believes that the concatenation is also fresh;
- **Believes(X,Said(Y,Conc(el1,el2))) $\Longrightarrow$ Believes(X,Said(Y,el1)) $\wedge$ Believes(X,Said(Y,el2))** - if X believes that Y said the concatenation of two elements, then he also believes that Y said both elements;
- **Believes(X,Says(Y,Conc(el1,el2))) $\Longrightarrow$ Believes(X,Says(Y,el1)) $\wedge$ Believes(X,Says(Y,el2))** - if X believes that Y recently said the concatenation of two elements, then he also believes that Y recently said both elements;

3.1.2.2 Cryptographic Rules

- **Sees(X,Sign(el,PrivKey(Y))) $\wedge$ Sees(X,PubKey(Y)) $\Longrightarrow$ Sees(X,el) $\wedge$ Believes(X,Said(Y,el))** - if X sees an element signed with another principal's private-key and if X sees the correspondent public-key then X can see the encrypted element and also knows who originally sent it;
- **Sees(X,Enc(el,PubKey(Y))) $\wedge$ Sees(X,PrivKey(Y)) $\Longrightarrow$ Sees(X,el)** - if X sees an element encrypted with another principal's public-key and if X sees the correspondent private-key then X can see the encrypted element. This happens when X and Y are the same principal and that represents the case where X sees its own private-key. This rule could also be applied in case the private-key was leaked or even shared. Sharing a private-key is not the most common nor the best solution for achieving delegation, but it is mentioned here and used in order to provide a simple example of delegation;
- **Sees(X,Enc(el,SharedKey k)) $\wedge$ Sees(X,SharedKey k) $\Longrightarrow$ Sees(X,el)** - if X sees an element encrypted with a shared-key and if X sees that same key then X can see the encrypted element;

3.1.2.3 Attacker Model Rules

All these rules contain also conditions to ensure the domain restriction. This is done by enforcing that the elements that are in the left side of the implications are part of one of the domains. In that way, we are able to prevent the infinite

concatenation or encryption of elements. Details on the implementation of the domain restriction can be seen in Section 7.3.3.

- **Sees(attacker,el1)** $\wedge$ **Sees(attacker,el2)** $\Longrightarrow$ **Sees(attacker,Conc(el1,el2))** - if the attacker sees two elements, he is also able to build their concatenation;
- **Sees(attacker,el)** $\wedge$ **Sees(attacker,PubKey(X))** $\Longrightarrow$ **Sees(attacker,Enc(el,PubKey(X)))** - if the attacker sees an element and a public-key, he can encrypt the element with that key;
- **Sees(attacker,el)** $\wedge$ **Sees(attacker,PrivKey(X))** $\Longrightarrow$ **Sees(attacker,Sign(el,PrivKey(X)))** - if the attacker sees an element and a private-key, he can sign the element with that key;
- **Sees(attacker,el)** $\wedge$ **Sees(attacker,SharedKey k)** $\Longrightarrow$ **Sees(attacker,Enc(el,SharedKey k))** - if the attacker sees an element and a shared-key, he can encrypt the element with that key;
- **Sees(attacker,el)** $\Longrightarrow$ **MsgSent(attacker,**X**,el)** - the attacker can send all the elements he sees to all the other principals;

3.2 The Underlying Engine

There are three parts that are used by the BAK tool to get the verification result: the built-in core of system rules R and the two required inputs: the model of the communication system (C) and the goal G that is verified by the tool (the goal has one or more security properties P). Having the model of the communications system (C) and the set of system rules (R), the tool tests each property P of the goal against the modeled communication system (C). The different parts of the BAK tool are defined in Figure 3.2.

Where C is a set of assertions that model the communication system, R is the set of inference rules that compose the core engine of the BAK tool, P is one of the predicates that compose the security goal G. The values used might either be variables (x) or constants (c), and *predicate* is one of the predicates presented in Figure 3.1.

$$
\begin{aligned}
System &::= C \wedge R \wedge G \\
C &::= \wedge_i(Assert_i) \\
R &::= Asserts_A \Rightarrow Asserts_B \\
G &::= \wedge_i((\neg)?P_i) \\
P &::= predicate(\vec{c}) \\
Asserts &::= Assert \wedge ... \wedge Assert \\
Assert &::= predicate(\vec{El}) \\
El &::= x|c
\end{aligned}
$$

Figure 3.2: BAK core assertions

3.2.1 Modeling and Verification with SMT

Having the model of the communication system C, then C_R is all the knowledge that one is able to infer from C using the inference rules R. A way of picturing such a system is shown in Figure 3.3.

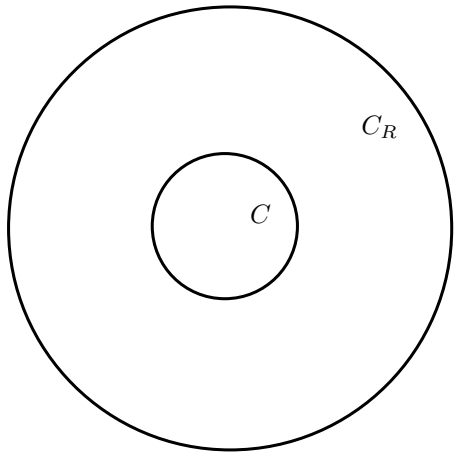


Figure 3.3: The knowledge inferred by communication system C using R.

Regarding C, we want to verify whether or not the system satisfies a given property P. In logic terms, this is stated as $C \wedge R \vdash P$. Indeed, we need to establish whether or not P is a logical consequence of the system, i.e., we need to establish that any instance of the system satisfies P. In order to make it simpler

to reason about the system, we require that P is an atomic positive assertion that could be derived by R. More specifically, P can either be a $Believes(.)$, used for verifying authentication, or a $Sees(.)$, used for verifying confidentiality.

Due to the way the system and the inference rules are designed, $C \wedge R$ does not contain any negative assertions. $C \wedge R$ can be seen as all the knowledge that can be derived from the communication system C using the inference rules R that contains the core engine of the BAK tool and which, as previously mentioned, contain rules that model both the attacker knowledge and the beliefs of the legitimate principals.

3.2.1.1 Verifying P in $C \wedge R$

We want to verify whether or not P is derived from C using R. That amounts to verify if $C \wedge R \vdash P$, which, in Figure 3.3, would be equivalent to verify that $P \in C_R$.

One way of verifying if P is derived from C using R is by checking $C \wedge R \wedge \neg P$ for satisfiability. If we have that

$$C \wedge R \wedge \neg P \; is \; SAT$$

which means that there is a model of $C \wedge R$ where $\neg P$ holds,i.e.:

$$C \wedge R \wedge \neg P \; is \; SAT \equiv \exists m \in M_{C \wedge R} : m \models \neg P$$

then we know that $P \notin C_R$ (P is not inferred from C using R). We know this because if it were the case that $P \in C_R$, the result would certainly be UNSAT, since having $P \wedge \neg P$ in a system would be a contradiction and would make the system unsatisfiable. If we have that

$$C \wedge R \wedge \neg P \; is \; UNSAT$$

which means that there is no model of $C \wedge R$ where $\neg P$ holds,i.e.:

$$C \wedge R \wedge \neg P \; is \; UNSAT \equiv \forall m \in M_{C \wedge R} : m \not\models \neg P$$

then it could either be that $P \in C_R$ (P is inferred from C using R) or that the modeling of the system led to a domain over-restriction. Therefore, if one wants to use the unsatisfiability result to prove that a property is derived from C using R, then one has to argue that the unsatisfiability is not due to the domain over-restriction.

It is worth noting that if one tries to check $C \wedge R \wedge P$ for satisfiability, the result will always be SAT due to the fact that, as previously discussed, $C \wedge R$ only derive positive assertions. Therefore, this SAT result will not distinguish between the case where P is derived from C using R from the case where P simply does not contradict any of the assertions derived by C using R (such contradiction would, in fact, never happen since $C \wedge R$ only contain positive assertions).

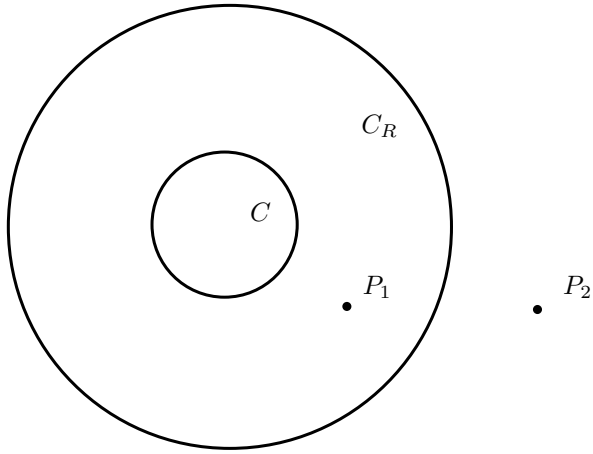


Figure 3.4: Security properties in the system.

Summarizing for the system being used, there are two different ways of proving security properties of the system, which are discussed bellow:

Proving that a property that models security is derived from C using R. This is the case of P_1 in Figure 3.4 and is related with reasoning about legitimate principals beliefs. For this case, there is a proof that P_1 is derived by $C \wedge R$ when $C \wedge R \wedge \neg P_1$ is unsatisfiable *and* it is shown that the unsatisfiability result was not due to the over-restriction of the domain cause by the analysis abstractions, i.e.:

$$(C \wedge R \wedge \neg P_1) \; is \; UNSAT \equiv_d C \wedge R \vdash P_1 \equiv P_1 \; holds \; in \; system \; C$$

where we have that $\equiv_d$ represents equivalence modulo domain-restriction, i.e., the equivalence exists over a specific size domain.

The domain size is relevant because if a limit is not set on the domain size, the tool might not terminate. For example, the result of concatenating two elements is another element. That concatenation element could then be concatenated again, and this chain of concatenations could happen forever and the analysis would, therefore, not terminate and that is why it is crucial to restrict the domain. On the other hand, precautions have to be taken when over-restricting the domain, i.e., for example, making the domain so small that the attacker is not able to perform the concatenation of two elements that is needed to perform an attack. Summarizing, the domain has to be limited so that the tool terminates and at the same time has to be big enough to avoid influencing the outcome of the analysis. The way this domain restriction is implemented in the BAK tool is detailed in Section 7.3.3.

Proving that a property that models the lack of security is not derived from C and R. This is the case of P_2 in Figure 3.4 and this kind of properties is related with the reasoning about the attacker knowledge. In this case, we have a proof that P_2 is not derived by $C \wedge R$ when $C \wedge R \wedge \neg P_2$ is satisfiable, i.e.:

$$(C \wedge R \wedge \neg P_2) \; is \; SAT \equiv C \wedge R \nvdash P_2 \equiv P_2 \; does \; not \; hold \; in \; system \; C$$

It is worth mentioning that Selinger [Sel03] developed an attacker-centric way of verifying protocols using a similar system. In his work, since he is only interested in reasoning about knowledge that might be derived by the attacker, his P is the attack being analyzed in the system. As it was shown above, if the result is SAT then one proves that the attack does not occur in the system. Since Selinger's work is only aimed at proving that the attack does not occur, assurances regarding the system that is being tested are only given when the analyzed system is satisfiable and, therefore, there is no need to reason about the domain restriction to defend the correctness of a protocol in that case.

3.2.1.2 Tool Outputs

When analyzing $C \wedge R \wedge (\neg P)$, the BAK tool does not only return the satisfiability analysis but also provides extra information that helps understand and analyze the obtained results. The kind of extra information will depend on whether the analysis result is satisfiable or unsatisfiable.

If the system and the goal being analyzed is satisfiable, then the tool also returns the model that satisfies the assertions. On the other hand, if system is unsatisfiable, the tool returns the unsatisfiability core, i.e., a small set of assertions that make the system unsatisfiable. This set is not guaranteed to be minimal but, in my experience, generally provides useful information regarding the system and the analysis result.

It is worth mentioning that extracting information from a satisfiable model is not as simple as extracting information from the unsatisfiable core. This is the case because the model tends to be complex and not easily readable. Analyzing and interpreting this model and providing more useful feedback to the user is part of future work.

3.3 The Modular Attacker

One of the the advantages of the BAK tool is the amount of flexibility provided for the modeling of the attacker capabilities. It is not only possible to directly model its capabilities in terms of constructing/reading different elements, but it is also simple to model whether or not the attacker is able to see the different exchanged messages. For example, one can model that the attacker is able to listen to the messages sent by Alice but not the ones sent by Bob. This can be specially useful when modeling cyber-physical systems, where an attacker does not always have access to the whole network. An example could be a situation like the one in Figure 3.5, where the attacker's physical access is restricted.

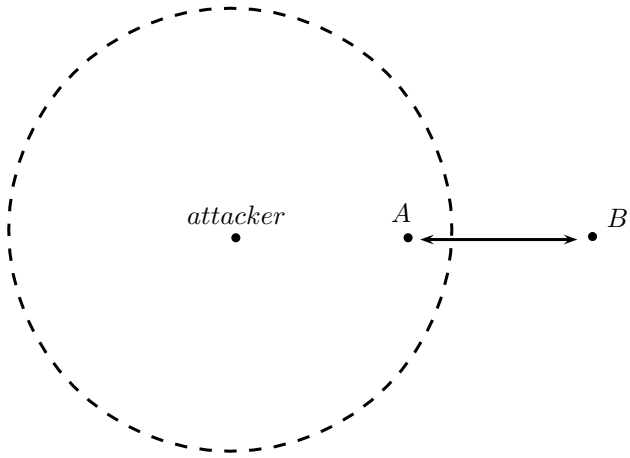


Figure 3.5: Attacker with limited physical access to the network.

In such a case, the standard rule:

$$\text{MsgSent(X,Y,el)} \Longrightarrow \text{Sees(Y,el)} \ \wedge \text{Sees(attacker,el)}$$

Would be replaced by the following two rules:

$$MsgSent(A, Y, el) \Longrightarrow Sees(Y, el) \wedge Sees(attacker, el)$$
$$MsgSent(B, Y, el) \Longrightarrow Sees(Y, el)$$

This way, it is modeled that the attacker can only see the messages sent by A and not the ones sent by B.

As for the situation where the attacker is sending the messages, the standard rule is the following:

$$\text{Sees(attacker,el)} \Longrightarrow \text{MsgSent(attacker,}X_i\text{,el)}$$

Which models the fact that the attacker is able to send all the elements he sees to all the principals. In this example, he is only able to send messages to A and not to B, so the rule is changed to the following rule:

$$\text{Sees(attacker,el)} \Longrightarrow \text{MsgSent(attacker,A,el)}$$

As intended, this rules defines that the attacker is only able to send messages to A.

Chapter 4

The Abstract Global Level

This level is the entry level for a system designer and its main goal is to provide the designer with a language that is simple and intuitive to use, while having the necessary tools to model the communication system under development.

4.1 The Language

As mentioned in Section 1.3.1, the modeling language at this level is similar to the Alice and Bob notation. The language has, however, an important extension: security modules. These security modules allow the developer to assign security assurances to the message elements being exchanged. This, similarly to CaPiTo (presented in Section 2.1.1), allows for the separation of concerns between the required types of security for the different message elements and the way that security is implemented.

The syntax of the language is shown is Figure 4.1. The system (*system*) is composed by a sequence of statements (*stm*), each of them representing a message (*msg*) being sent from one principal to another. A message is a sequence of elements (*el*) that are either plain-text or security modules.

system	::=	**stm;** | **system stm;**
stm	::=	**principal** → **principal : msg**
principal	::=	**string**
msg	::=	**el** | **msg, el**
el	::=	**string** | **secModule**
secModule	::=	**secAssurance**(**msg**)
secAssurance	::=	Auth | StrongAuth | Conf | Sec | StrongSec

Figure 4.1: Syntax of the Abstract Global level language.

4.2 The Security Modules

As mentioned above, the most important elements in this level are the security modules, which allow for the modeling of security assurances of the exchanged data. Firstly, an explanation of the different security properties is given, then a simple overview of different modules is shown in Table 4.1 and a more detailed description of their semantics is given in Section 4.2.2.

4.2.1 Authentication, Confidentiality, and Security

Authentication is a highly discussed property not only in network security but also in many other areas. There are, in fact, different notions of authentication, mostly regarding its precise meaning and which exact security properties must be verified in order to achieve authentication. Gollmann [Gol96] describes his notion of authentication by having it composed by two parts: message origin authentication and replay prevention. Having message origin authentication, one is able to achieve what can be called strong authentication by adding a mechanism that prevents replay attacks, i.e., a mechanism that does not allow an attacker to impersonate the agent being authenticated by re-using the exchanged messages. This can be implemented by adding freshness guarantees to the exchanged messages, for example, by using a time-stamp or a challenge sent by the authenticator. Another way of defining authentication, similar to the one provided by Gollmann, but renamed in a way that better aligns with the rest of the work presented here, is:

$$\textit{Strong Authentication} = \textit{Authentication} + \textit{Freshness Guarantees}$$

Mödersheim and Viganò [MV09b] also present a notion of authentication when they introduce the security properties of different communication channels (introduced in Section 2.4). In general terms, the authors' description of authentication is similar to Gollmann's description: what Mödersheim and Viganò call authentication is what Gollmann calls message origin authentication and what is called authentication in this work. As for Gollmann's authentication, Mödersheim and Viganò call it authentication with freshness and that is called strong authentication in this work.

One important addition to authentication made by Mödersheim and Viganò is the use of a receiver identifier in the authenticated message. They argue [MV09c] that authentication should also ensure that the authenticated message is intended to a specific recipient. Adding the receiver identifier to an authenticated message has a very small cost and it aids on the prevention of replay attacks. A simple example of that is that if there was a message exchange where Alice is sending an authentic message m without adding the receiver identifier:

$$A \rightarrow B : \{m\}_{privKey(A)}$$

it would then be possible for a dishonest Bob to send that message to Carsten:

$$B \rightarrow C : \{m\}_{privKey(A)}$$

and that would result in Carsten thinking that he was receiving that message directly from Alice.

If, on the other hand, we have Alice also sending the receiver identifier together with the message:

$$A \rightarrow B : \{B, m\}_{privKey(A)}$$

then if Bob replays that message to Carsten:

$$B \rightarrow C : \{B, m\}_{privKey(A)}$$

Carsten will know that the message was indeed sent by Alice at some point but not originally to him but to Bob.

The compositionality of security modules (or channels) is investigated by Mödersheim and Viganò. They defend that compositionality might not preserve the different security properties and they provide a set of restrictions under which compositionality does preserve the security properties. The work presented here does not deal directly with compositionality but, by having verification tools in the concrete levels of the frameworks, it allows for the verification of the security properties after the security modules are replaced by their implementations and, therefore, with the composed implementation of the security modules.

Another simple mechanism, also mentioned by Mödersheim and Viganò, that helps prevent possible compositionality issues and type-flaw attacks is the usage of tags on the different messages. Having a tag that helps identifying the original intent of a message prevents some cases where the attacker could replay a message in a different context. This mechanism is a small cost to pay to add some extra protection against type-flaw attacks. Therefore, tags are also used in the cryptographic implementation of the different modules in this project. $authTag$ (or aT for brevity) is used for authentication and $confTag$ (or cT for brevity) for confidentiality. There is no tag for security because, as it will be shown below, the implementation of the security module uses authentication and confidentiality.

Yet another important work on the definition of authentication and its properties is given by Lowe [Low97]and it is also possible to describe the aforementioned authentication and strong authentication with Lowe's definitions of authentication.

Authentication can be characterized as Lowe's non-injective agreement while strong authentication as Lowe's (injective) agreement. What Lowe means by injective and non-injective agreement is related with the communication being susceptible to replay attacks. Injective agreement means that there is a one-to-one correspondence between the initiation of the authentication communication on the sender's side (the sending of an authenticated message) and the ending of that communication on the receiver's side (the reception and interpretation of the authenticated message). This means that no replay attacks can occur in an injective agreement. In case replay attacks might occure, that amounts to a correspondence of one-to-many, which is what Lowe calls non-injective agreement.

Confidentiality is a less controversial security property and there is a general consensus on what it means: that only the intended receiver of the message is able to read that message. In other words, when Alice sends a confidential message to Bob, she knows that only Bob is able to read that message and when Bob receives that message, he also knows that he is the only one able to read that message.

As for security, it can be seen as the combination of authentication and confidentiality, i.e., when a secure message is sent from Alice to Bob, Bob knows that Alice was the originator of the message and they both know that only Bob can read the message. One can then have security or strong security depending on the used authentication variation: without or with freshness guarantees, respectively.

The security modules, their informal meanings, and a possible cryptographic implementation are shown in Table 4.1. A more formal description of the security modules is presented in Section 4.2.2. There are several possible ways of implementing the security modules, but the chosen implementation uses cryptographic mechanisms and a Public-Key Infrastructure, which enables the use of cryptography with public-/private-key pairs. The main reasons behind this choice are that it allows for a simple implementation with few exchanged messages and its simplicity also helps on further understanding the semantics of the security modules.

It is worth noting the presence of N in the strong authentication and strong security cryptographic implementations in Table 4.1. That N is used in a way that guarantees the freshness of the message to the recipient (B) of the message. That could be implemented by having N being a time-stamp that B can verify or by implementing an extra message exchange where B sends a freshly generated element to A prior to the message sent from A to B and, therefore, B is be able to check that the message is fresh since it contains the previously sent freshly generated element. This possible implementation for the StrongAuth module is shown bellow:

$$B \rightarrow A : N$$
$$A \rightarrow B : \{aT, B, N, m\}_{privKey(A)}$$

4.2.2 Semantics of the Security Modules

To define the semantics of the security modules at the Abstract Global level, the BAN logic was extended with a principal variable *attacker*, which is used to represent am explicit attacker.

Before defining the semantics of the security modules, it is important to expand on the use of integrity in GSD. Integrity can have different meanings depending on the field it is being used in, and can even have slightly different definitions in the same field. For this work, integrity is considered to mean that a message is not corrupted over time or in transit [CHL+00].

Security Modules	Description	Cryptographic implementation
None	represents the case that data is sent in plain-text	$A \rightarrow B : m$
Authentication	data is transmitted in a way that allows the receiver to identify the original sender	$A \rightarrow B : \{aT, B, m\}_{privKey(A)}$
Strong Authentication	similar to the Authentication module, but also provides freshness guarantees	$A \rightarrow B : \{aT, B, N, m\}_{privKey(A)}$
Confidentiality	data is transmitted in a way that only allows the intended receiver to read it	$A \rightarrow B : \{cT, m\}_{pubKey(B)}$
Security	conjugation of the Weak Authentication and the Confidentiality modules	$A \rightarrow B : \{cT, \{aT, B, m\}_{privKey(A)}\}_{pubKey(B)}$
Strong Security	similar to the Weak Security module, but also provides freshness guarantees. It is a conjugation of the Authentication and the Confidentiality modules	$A \rightarrow B : \{cT, \{aT, B, N, m\}_{privKey(A)}\}_{pubKey(B)}$

Table 4.1: The Security Modules

For message exchange, integrity is guaranteed when the contents of a message cannot be changed in transit without changes being instantly obvious to the recipient of that message. In the GSD framework, integrity is assumed in all the messages that are exchanged. This can be implemented by, for example, sending a signed digest of the message together with the full message. With integrity, the following rule is present in the system:

$$\frac{X \rightarrow Y : m}{\begin{array}{l} Y \ sees \ m, \\ attacker \ sees \ m \end{array}}$$

Another important notation is *represents*. When having X *represents* Y, it means that X is able to act as Y, i.e., to represent it. It means that X is trusted by X or that acquired sufficient private knowledge that enables them to authenticate as X. This way of modeling the situation where principals are being trusted and are allowed to represent other principals is similar to the one used in the BAN logic. It is an elegant and simple solution that, at the same time, gives the power and flexibility for reasoning about delegation and trust issues. In Section 7.3.4, a more detailed description of how this rule is implemented in the GSD framework prototype is given.

It is also worth noting that in the security modules going to be presented now, what is being considered the authentication of the message and not of the sender of the message and that is the reason why the receiver identifiers are not used in these modules. In other words, we want to have assurances regarding which principal said a message (i.e., the original sender of the message), but not assurances regarding the last sender of the message (i.e., if an authenticated message is forwarded by a principal, the assurances are still concerning the original sender of the message and not concerning the principal who forwarded it).

4.2.2.1 Authentication

The rule for the authentication module specified that when a principal Z sees $Auth(X, w)$, then Z knows that the message was originally sent by X, but knows nothing about the freshness of the message:

$$\frac{Z \ sees \ Auth(X, w)}{Z \ believes \ X \ said \ w}$$

4.2.2.2 Strong Authentication

As previously mentioned, having authentication ($Auth(X, w)$) without any assurances in terms of freshness does not provide any guarantees regarding when principal X sent the message and is, therefore, susceptible to replay attacks — where a principal that got access to a message previously sent is able to resend it. That is the reason for the existence of the strong authentication module (called $StrongAuth$) that also provides freshness guarantees:

$$\frac{Z \ sees \ StrongAuth(X, w)}{\begin{array}{l} Z \ believes \ X \ said \ w, \\ Z \ believes \ fresh(w) \end{array}}$$

It is worth nothing the belief $Z \ believes \ fresh(w)$. When implementing the modules, one has to make sure that guarantees regarding freshness are implemented, for example, using timestamps or exchange of nonces to provide that freshness.

Having the following rule that models the influence of freshness:

$$\frac{Z \ believes \ X \ said \ w, \ Z \ believes \ fresh(w)}{Z \ believes \ X \ says \ w}$$

the reasoning associated with the $StrongAuth$ module can then be completed:

$$\frac{\begin{array}{l} Z \ sees \ StrongAuth(X, w) \\ \hline Z \ believes \ X \ said \ w, \\ Z \ believes \ fresh(w) \end{array}}{Z \ believes \ X \ says \ w}$$

4.2.2.3 Confidentiality

The rule for the Confidentiality module specifies that a message that is confidential to Y can only be read by him, or by another principal that represents Y:

$$\frac{Z \textit{ sees } Conf(Y, w), \quad Z \textit{ represents } Y}{Z \textit{ sees } w}$$

It is important to note that there are two situations where a principal (including the attacker) is able to see w: if the principal has been previously authorized by Y to represent him or if the attacker is able to derive sufficient knowledge that enables him to represent Y.

4.2.2.4 Security

Security can be seen as the conjugation between authentication and confidentiality. Therefore, its rule is the following:

$$\frac{Z \textit{ sees } Sec(X, Y, w), \quad Z \textit{ represents } Y}{\begin{array}{l} Z \textit{ sees } w, \\ Z \textit{ believes } X \textit{ said } w \end{array}}$$

4.2.2.5 Strong Security

Similarly to the Authentication module, $Sec(X, Y, w)$ does not provide any guarantees regarding when principal X sent the message and is also susceptible to replay-attacks. That is why the security module *StrongSec* was created, to add freshness guarantees to the security module:

$$\frac{\begin{array}{l} Z \textit{ sees } StrongSec(X, Y, w), \quad Z \textit{ represents } Y \end{array}}{\begin{array}{l} Z \textit{ sees } w, \\ Z \textit{ believes } X \textit{ said } w, \\ Z \textit{ believes } fresh(w) \end{array}}$$
$$\begin{array}{l} Z \textit{ sees } w, \\ Z \textit{ believes } X \textit{ says } w \end{array}$$

Conjugation of modules. As previously mentioned, the security module is a conjugation of authentication and confidentiality:

$$X \to Y : Sec(m) \Leftrightarrow X \to Y : Conf(Auth(m))$$

By looking at this composition of authentication and confidentiality to obtain security, one could ask why the modules are composed as $Conf(Auth(m))$ and not as $Auth(Conf(m))$. The reason for this is that, if the composition is performed as $Auth(Conf(m))$ then it would not provide the necessary beliefs for a security module. It is possible to see that by looking at an example where X sends to Y the message $Auth(Conf(m))$. Then Y sees $Auth(A, Conf(B, m))$ and the possible derived beliefs would be:

$$\frac{\frac{\frac{X \rightarrow Y : Auth(Conf(m))}{Y \ sees \ Auth(X, Conf(Y, m))}}{\begin{array}{l} Y \ believes \ X \ said \ Conf(Y, m) \\ Y \ represents \ Y \end{array}}}{\begin{array}{l} Y \ sees \ m, \\ Y \ believes \ X \ said \ Conf(Y, m) \end{array}}$$

And these beliefs are not the desired beliefs for a security module (as shown is Section 4.2.2.4). In contrast to that, if the conjugation of the authentication and the confidentiality modules is done as $Conf(Auth(m))$, that results in the following beliefs:

$$\frac{\frac{\frac{X \rightarrow Y : Conf(Auth(m))}{\begin{array}{l} Y \ sees \ Conf(Y, Auth(X, m)), \\ Y \ represents \ Y \end{array}}}{Y \ sees \ Auth(X, m)}}{\begin{array}{l} Y \ sees \ m, \\ Y \ believes \ X \ said \ m \end{array}}$$

Which as one can see are the desired beliefs for a security module that are shown is Section 4.2.2.4). The composition of the strong authentication and confidentiality is performed in a similar way in order to achieve the strong security module.

4.2.3 The Contracts

Another important component of the GSD framework that is strongly connected with the Abstract Global level is the use of contracts, which are attached to the

security modules presented above and, therefore, model the outcome of using the different security modules. One could say that contracts are the implementation of the security modules semantics presented above. This enables the verification of the different implementations of the security modules by checking that the security properties specified in the contracts are achieved by the used implementations.

In other words, the contracts impose the beliefs that the principals must possess after running the implementation of the respective security module. The contracts define not only the beliefs of the principals after a normal run of the modeled protocol, but also the beliefs of the attacker and what he is able to see.

More information regarding the implementation and usage of these contracts in the GSD framework prototype is given in Chapter 7. Details of their use with OFMC are given in Section 7.2.1, and details on their use in the BAK tool are given in Section 7.3.1.

4.3 Abstract Global Level outputs

Even though the modeling language at this level is considerably abstract, it is possible to perform some formal verification with OFMC already at this level. That is advantageous since the earlier any security flaws can be found in the design phase, the less expensive they will be to correct.

4.3.1 The OFMC Analysis

The OFMC analysis at this level is possible due to the similarities between the security modules and the channels used in the AnB• language used by OFMC. The model of the system described in language of the Abstract Global level is translated into the AnB• language, so that it can be verified by OFMC. The translation is shown in Table 4.2. It is worth noting the use of nonces in the cases of the *StrongAuth* and *StrongSec*. This happens because the current version of OFMC (2012c) does not yet support the specification of the fresh authentication and secure channels and, for that reason, the strong authentication and strong security modules were implemented by introducing a nonce in the authentication and secure channels, respectively. An alternative to the chosen implementation could be to have the specification translated into AnBx [BM11], which, as was mentioned in Section 2.4, allows the modeling of authentication and secure channels with freshness guarantees and the translation from the se-

curity modules to the AnBx channels could be performed directly, i.e., without having to recur to nonces. AnBx could then be translated to AnB, which could then be used with OFMC.

Using OFMC's analysis at this point of the system development does not provide a great amount of information, since the system designer only specified the desired security properties for the different messages, but it can be used as a sanity check for the given system model, allowing the designer to confirm that the communication system, and the security properties, were modeled correctly.

Security property	Specification with security modules	Specification with channels
None	$A \rightarrow B : msg$	$A \rightarrow B : msg$
Auth	$A \rightarrow B : Auth(msg)$	$A \bullet \rightarrow B : msg$
StrongAuth	$A \rightarrow B : StrongAuth(msg)$	$B \rightarrow A : nonce$ $A \bullet \rightarrow B : nonce, msg$
Conf	$A \rightarrow B : Conf(msg)$	$A \rightarrow \bullet B : msg$
Sec	$A \rightarrow B : Sec(msg)$	$A \bullet \rightarrow \bullet B : msg$
StrongSec	$A \rightarrow B : StrongSec(msg)$	$B \rightarrow A : nonce$ $A \bullet \rightarrow \bullet B : nonce, msg$

Table 4.2: Translation of security modules into AnB channels.

4.4 The Message Board

In the message board example, security modules are used to model the authenticated message sent by the different users to the `Message Board (MB)` as shown in Listing 4.1.

```
Alice → MB : StrongAuth(msg1);
Bob → MB : Conf(msg2);
Carsten → MB : Sec(msg3);
```

Listing 4.1: Specification of the message board example.

The AnB code that is automatically generated from the specification of the system presented in Listing 4.1 is shown in Listing 4.2. The goals are automatically generated by using the contracts of the used security modules in the Abstract Global level specification. More details regarding the generation of the goals are given in Section 7.2.1.

```
Protocol: manual
Types:
    Agent alice, bob, carsten, mb;
    Number empty, ...;
    Function pk ...
Knowledge:
    alice: ...;
    bob: ...;
    carsten: ...;
    mb: ...
Actions :
    mb → alice : N200
    alice *→ mb : N200,msg1
    bob →* mb : msg2
    carsten *→* mb : msg3
Goals :
    mb authenticates alice on msg1
    msg2 secret between bob,mb
    mb weakly authenticates carsten on msg3
    msg3 secret between carsten,mb
```

Listing 4.2: Specification of the exchanged messages from the message board example.

In order to being able to use the automatically generated code (shown in Listing 4.2) with OFMC, one needs to manually provide some extra information: the elements of the exchanged messages have to be declared in the `Types` section and the initial knowledge of the different principals has to be modeled in the `Knowledge` section. The last change that needs to be performed is the eventual introduction of dummy messages (which by themselves are not relevant for the system) in the `Actions` section in case the modeled communication is not performed in a way that resembles the use of a communication system based on a token. For example, that is not the case in the automatically generated code shown in Listing 4.2 and that is why dummy messages were introduced, as it can be seen in lines 14 and 16 of Listing 4.3. The added information could, in fact, be automatically derived from the system model and it is part of future work in the development of the GSD framework prototype, as mentioned in Section 9.1.

```
1  Protocol : manual
2  Types :
3      Agent alice,bob,carsten,mb;
4      Number msg1,msg2,msg3,N200, authTag, confTag, empty;
5      Function pk
6  Knowledge :
7      alice: alice, mb, inv(pk(alice)), msg1, pk(mb),authTag, confTag
           ;
8      bob: bob, mb, msg2, pk(mb),authTag, confTag;
9      carsten: carsten, msg3, inv(pk(carsten)), mb, pk(mb), authTag,
           confTag;
10     mb: mb, alice, carsten, inv(pk(mb)), pk(alice), pk(carsten),
           authTag, confTag, empty
11 Actions :
12     mb → alice : N200
13     alice *→ mb : msg1
14     mb → bob : empty
15     bob →* mb : msg2
16     mb → carsten : empty
17     carsten *→* mb : msg3
18 Goals :
19     mb authenticates alice on N200,msg1
20     msg2 secret between bob,mb
21     mb weakly authenticates carsten on msg3
22     msg3 secret between carsten,mb
```

Listing 4.3: Completed specification of the exchanged messages from the message board example.

The completed code is shown in Listing 4.3 and the result of the OFMC analysis of that code is shown in Listing 4.4, which reports that no attack was found (`NO_ATTACK_FOUND`). This means that the usage of the chosen security channels for the message board example provide the required security properties for the exchanged messages. There are, in fact, two results being displayed together, the first result (lines 1-11) for the default OFMC execution that performs a fixed-point analysis with unbounded number of sessions. The second result (lines 13-27) is the output of running OMFC in the model-checking mode, where OFMC performs a model-checking analysis of the system with a bounded number of sessions. The GSD framework is currently requesting OFMC to check up to 4 sessions, so that the analysis does not take too long, but that number can be set higher if ones wishes so. It is also worth noting that the description of the goals specified in lines 7 and 19 are hard-coded in the tool output and do not depend on the goals being checked.

```
Result for default OFMC run:
% Open-Source Fixedpoint Model-Checker version 2012c
INPUT
    tmp1243139937.AnB
SUMMARY
  NO_ATTACK_FOUND
GOAL:  honest-weak authentication and secrecy
DETAILS
  UNBOUNDED_NUMBER_OF_SESSIONS, TYPED_MODEL, FREE ALGEBRA
BACKEND OFMC
COMMENTS It is recommended to run with --classic option as well (
    see manual).

Result for classic OFMC run:
% Open-Source Fixedpoint Model-Checker version 2012c
INPUT
    tmp1243139937.AnB
SUMMARY
  NO_ATTACK_FOUND
GOAL:  as specified
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
BACKEND OFMC
STATISTICS
  TIME 46 ms
  parseTime 0 ms
  visitedNodes: 79 nodes
  depth: 8 plies
```

Listing 4.4: OFMC analysis result of the message board example.

Chapter 5

The Concrete Global Level

The specification of the system at this level is more concrete (hence the level name) than the specification in the Abstract Global level and it is obtained by the automatic guided translation from the Abstract Global level that is described in Section 5.1.

At the Concrete Global level, the GSD framework interfaces with the Beliefs and Knowledge tool (Section 5.3.1) and OFMC (Section 5.3.2). Furthermore, as it is shown in the use case in Chapter 8, when a system is already implemented it can be specified directly at this level and verified by using the interface with the verification tools.

5.1 Unfolding the Security Modules

To translate the system specification from the Abstract Global level to the Concrete Global level, the security modules present in the Abstract Global level are unfolded. This unfolding is done by applying one of the different plugins for each security module. These plugins represent implementations of the correspondent security module using, for example, the Transport Layer Security (TLS) protocol, WS-Security (which adds security to the SOAP web services),

or a Public-Key Infrastructure (PKI). As mentioned when introducing the contracts in Section 4.2.3, it is possible to verify the chosen implementation for a security module by checking that it satisfies the respective contract.

The syntax for this level is shown in Figure 5.1. As one can see, the syntax of this level is similar to the one of the Abstract Global level with the main difference being the absence of the security modules and the addition of cryptographic primitives that allow the modeling of the different implementation possibilities.

system	::=	**stm;** | **system stm;**
stm	::=	**principal** → **principal** : **msg**
principal	::=	string
msg	::=	**el** | **msg, el**
el	::=	**string** | **encModule** | **hash**
hash	::=	Hash(**msg**)
encModule	::=	Enc(**msg**,**key**)
key	::=	PrivKey(**principal**) | PubKey(**principal**) | SharedKey

Figure 5.1: Syntax of the Concrete Global level

5.1.1 The Plugins

The plugins are an important part of the Guided System Development framework since they are used to implement the security modules.

A big advantage of the use of the plugins is the flexibility they provide on the choice of the implementations for each of the desired security properties, i.e., the system designer is able to choose different implementations that are guaranteed to achieve the security properties modeled in the system description at the Abstract Global level. Another important advantage of the plugins is that they enable a more dynamic approach to building secure systems since the plugins make it simple to provide new implementations of the different security properties or also implementations of new security properties. A brief overview of possible security mechanisms and a suggestion of their implementation as plugins is given in the sections below. These are to be viewed as suggestions and they should be further investigated before being implemented, which is part of future work.

5.1.1.1 The PKI Plugin

A possible approach is to create a set of plugins that use a Public-Key Infrastructure (PKI) to implement the different security modules. A Public-Key Infrastructure uses Certificates and Certificate Authorities (CA) to bind the public-key to a specific principal. That public-key is part of a key pair that also contains a private-key, which is (usually) secret. This public-/private-key pair has characteristics that make it useful when encrypting data: a message encrypted with one of the keys can only be decrypted by using the other key in the respective pair, which makes PKI useful for achieving authentication and confidentiality as it is shown bellow.

This approach provides a lightweight and simple message exchange with a small overhead regarding the exchanged data.

The way to implement the different security modules (shown in Table 4.1) is summarized here in Figure 5.2.

- Authentication: authentication tag, receiver's id, and message are encrypted with sender's private key;
- Strong Authentication: authentication tag, receiver's id, message, and fresh element are encrypted with sender's private key;
- Confidentiality: confidentiality tag and message are encrypted with the receiver's public key;
- Security: authentication tag, receiver's id, and message are encrypted with sender's private key and then encrypted (together with the confidentiality tag) with receiver's public key;
- Strong Security: authentication tag, receiver's id, message, and fresh element are encrypted with sender's private key and then encrypted (together with the confidentiality tag) with receiver's public key;

Figure 5.2: Description of the PKI plugin.

The actual implementation of the different modules using PKI is shown in Figure 5.3. In this implementation, an encryption scheme that supports private-key encryption as a method of sign as message is used. A encryption scheme that allows that is, for example, RSA [RSA78].

- Authentication: $A \rightarrow B : \{aT, B, m\}_{privKey(A)}$
- Strong Authentication: $A \rightarrow B : \{aT, B, N, m\}_{privKey(A)}$
- Confidentiality: $A \rightarrow B : \{cT, m\}_{pubKey(B)}$
- Security: $A \rightarrow B : \{cT, \{aT, B, m\}_{privKey(A)}\}_{pubKey(B)}$
- Strong Security: $A \rightarrow B : \{cT, \{aT, B, N, m\}_{privKey(A)}\}_{pubKey(B)}$

Figure 5.3: Implementation of the security modules with PKI.

5.1.1.2 The Transport Layer Security Protocol

The Transport Layer Security protocol (TLS) [DR08] is one of the most commonly used security protocol on the Internet. It allows for the establishment of a secure connection between the client and the server also using a Public-Key Infrastructure (PKI). TLS has two main modes that determine the characteristics of the connection. One mode is called *TLS with server side authentication*, where only the server is authenticated by sending its credentials when establishing the connection, and the other mode is called *TLS with mutual authentication*, where both the client and the server are authenticated using their own PKI credentials when establishing the connection. The authentication is performed by verifying that the public-/private-key pair used in the communication belongs to the intended entity. That is done by having Certificate Authorities (CA) issue certificates for the public-key of those key pairs.

A common practice in the Internet is to use TLS only with server side authentication and, after establishing the connection, having the client authenticate with the server by transmitting its username and password. The implementation of the security modules with TLS are shown in Figure 5.4.

- Authentication: not available with TLS;
- Strong Authentication: not available with TLS;
- Confidentiality: TLS with server side authentication - since after setting this communication channel there are guarantees regarding the server's identity, the information sent by the client can only be read by the server (i.e., the intended recipient);
- Security: not available with TLS;
- Strong Security: TLS with mutual authentication - in this communication channel, the identities of the client and the server are guaranteed and, therefore, the information sent over the channel can only be read by the server (i.e., the intended recipient) and it has assurances regarding the sender of the information;

Figure 5.4: Suggestion for implementing the plugins with TLS.

5.1.1.3 WS-Security

WS-Security [NKHBM04] is an end-to-end solution to add security to the SOAP web services. It does so not by proposing new security methods and techniques but by leveraging current security technology and describing how to apply the latter to the SOAP web services in order to achieve the required security properties, i.e., WS-Security is a framework to embed existing security mechanisms into a SOAP message in a way that is independent from the used transport technology.

WS-Security addresses message authentication (with freshness guarantees), message integrity (using digital signatures) and message encryption. A possible implementation of the plugins using these mechanisms is shown in Figure 5.5.

- Authentication: not available with WS-Security;
- Strong Authentication: WS-Security with message authentication - the message authentication provides assurances regarding the identity of the sender of the message;
- Confidentiality: WS-Security with message encryption - the message encryption provides assurances regarding who can read the message;
- Security: not available with WS-Security;
- Strong Security: WS-Security with message authentication and encryption - the message authentication and encryption provide assurances regarding the assurances regarding the identity of the sender and regarding who is able to read the message;

Figure 5.5: Suggestion for implementing the plugins with WS-Security.

5.1.1.4 IPSEC

IPsec [KS05] is a suite of protocols for securing network connections that operates in a lower layer of the TCP/IP model, i.e., in the Internet Layer. It has an extensive list of settings and part of it is shown below:

- AH vs ESP
  - AH - provides sender authentication and message integrity;
  - ESP - provides sender authentication, encryption, and message integrity;
- Tunnel Mode vs Transport Mode
  - Tunnel Mode - in this mode, the entire IP packet is encapsulated;
  - Transport Mode - in this mode, only the IP payload is encapsulated;
- Encryption schemes options
  - DES, 3DES, and AES among others;
- Hashing schemes options
  - MD5 and SHA-1 among others;
- IKE vs manual keys

  - It is possible to choose how the required keys are distributed prior to IPsec;

- Main mode vs Aggressive mode
  - Main mode
    - This mode is more secure and six messages (packets) are exchanged;
  - Aggressive mode
    - This mode is more time efficient but less secure, since only three messages (packets) being exchanged with some information being sent in plain-text;

Having these settings in consideration, a possible implementation of the plugins with IPsec is shown in Figure 5.6.

- Authentication: not available with IPsec;
- Strong Authentication: IPsec with AH mode - as shown above, AH mode provides message authentication, i.e., guarantees regarding the identity of the sender;
- Confidentiality: not available with IPsec;
- Security: not available with IPsec;
- Strong Security: IPsec with ESP mode - as shown above, ESP mode provides sender authentication and encryption, which gives guarantees regarding the identity of the sender and regarding who is able to see the message;

Figure 5.6: Suggestion for implementing the plugins with IPsec

5.2 The Message Board

For our example system, the modules could be implemented with several of the plugins presented above, but below we show the implementation of the modules with the PKI plugin that was implemented in the prototype tool.

5.2.1 Using PKI

The result of applying the PKI plugin to the message board example is shown in Listing 5.1.

```
MB → Alice : N200;
Alice → MB : Enc((authTag, MB, N200, msg1), PrivKey(Alice));
Bob → MB : Enc((confTag, msg2), PubKey(MB));
Carsten → MB : Enc((confTag, Enc((authTag, MB, msg3), PrivKey(
    Carsten))), PubKey(MB));
```

Listing 5.1: The implementation of the message board example using the PKI infrastructure.

5.3 Concrete Global Level Outputs

In this section, the formal methods tools that GSD currently interfaces with at the Concrete Global Level are presented. As mentioned when presenting the Beliefs and Knowledge (BAK) tool in Section 3, the BAK tool was developed specifically for the GSD framework and the frameworks connects with it from this level. In addition, the GSD framework also connects with OFMC from this level.

5.3.1 The BAK tool Analysis

Listing 5.2 shows the model of the the message board example implemented with PKI (as shown in Listing 5.1) in SMT-LIB2, a standard format accepted as input by several SMT solvers, including Z3, the engine used by the BAK tool.

```
(assert (Believes MB (Fresh N200)))
(assert (MsgSent MB Alice N200))
(assert (MsgSent Alice MB (PubKeySign authTag (Conc N200 msg1) (
    PrivKey Alice))))
(assert (MsgSent Bob MB (PubKeyEnc confTag msg2) (PubKey MB)))
(assert (MsgSent Carsten MB (PubKeyEnc confTag ((PubKeySign authTag
    msg3) (PrivKey Carsten)) (PubKey MB))))
```

Listing 5.2: An authenticated message of the example system modeled in SMT-LIB2.

The result of applying different tests to the model of the example system is shown in Table 5.1. These goals are the contracts of the different security modules as explained in Section 7.3.1.

Test	Result	Result Interpretation
not (Believes MB (Says Alice msg1))	unsat	MB believes that Alice recently said msg1
not (Sees MB msg2)	unsat	Mb sees msg2
not (Sees attacker msg2)	sat	attacker does not see msg2
not (Believes MB (Said Carsten msg3))	unsat	MB believes that Carsten said msg3 at some point
not (Sees Mb msg3)	unsat	MB sees msg3
not (Sees attacker msg3)	sat	attacker does not see msg3

Table 5.1: Output of the BAK tool for the example system.

In Table 5.1, only the sat/unsat result is shown, but the BAK tool does return more information. For example, the complete result of testing the two goals related with msg2 — second and third line of Listing 5.1 — is shown in Listing 5.3. The first line tells us that the system model together with the first negated goal is unsatisfiable, which means that the property without the negation holds, fact that is mentioned in the Result Interpretation column of Table 5.1. The second line of the output is the unsatisfiability core returned by Z3. It displays the names of the rules and assertions that are used to arrive at one unsatisfiability core (not necessarily the minimal one). This result tells us that `MB` sees `msg2` because of a message that was sent (`msgSent`) and because he was able to decrypt that message (`decOfPubKeyEnc`) due to the fact that it knows its own private-key (`MBseesItsPrivKey`). Therefore, we know that `MB` received a message, decrypted it with its private-key which allowed him to see `msg2`. Line 3 of Listing 5.3 shows the result for the second test related with `msg2`. It would also be possible to show a satisfiable model, but it was left out due to its size. An interesting possibility for future work is the extraction of information from that satisfiable model.

```
unsat
(MBseesItsPrivKey msgSent decOfPubKeyEnc)
sat
```

Listing 5.3: Complete output of the BAK tool for one of the tests from example system.

5.3.2 The OFMC Analysis

In Listing 5.4, the automatically generated output of AnB code from the Concrete Global level is shown. It is the skeleton of the file used as input to OFMC. As mentioned in Section 4.4, the goals in Listing 5.4 are automatically generated by using the contracts of the used security modules in the Abstract Global level specification. More details are given in Section 7.2.1.

It is also worth noting that, both in here and in the AnB code generated from the Abstract Global level (Section 4.4), the names of the principals are in lower case. That is an important detail, since OFMC considers principals whose name starts in lower case as honest principals and in capital case as dishonest principals. The main reason for the decision of having the principals in lower case, and therefore honest principals, is that not much information can be taken from checking a message for confidentiality when the principal that is sending that message encrypted in a way that would preserve confidentiality can also act dishonestly and send that same message in plain-text. Having the principals in capital case, specially when verifying authentication properties is, however, an interesting possibility and would, therefore, be an interesting subject of future work.

```
Protocol: manual
Types:
   Agent alice, bob, carsten, mb;
   Number empty, ...;
   Function pk ...
Knowledge:
   alice: ...;
   bob: ...;
   carsten: ...;
   mb: ...
Actions :
   mb → alice : N200
   alice → mb : {authTag,MB, N200,msg1}inv(pk(alice))
   bob → mb : {confTag,msg2}pk(mb)
   carsten → mb : {confTag,{authTag,MB,msg3}inv(pk(carsten))}pk(mb)
Goals :
   mb authenticates alice on msg1
   msg2 secret between bob,mb
   mb weakly authenticates carsten on msg3
   msg3 secret between carsten,mb
```

Listing 5.4: The automatically generated AnB code of message board example.

In Listing 5.5 one can see the complete AnB file that is used as input to OFMC. In this case, one needs to add in section `Types` the declaration of the messages being exchanged (present in line 4) and the used functions in line 5, and in section `Knowledge` the initial knowledge of the principals (modeled in lines 7

to 10). Furthermore, and also as mentioned in Section 4.4, one also needs to introduce dummy messages in the `Actions` section. Most of this information can be extracted from the system model present in the GSD framework. That is, in fact, planned for future work on the further development of the framework prototype.

```
Protocol : manual
Types :
 Agent alice,bob,carsten,mb;
 Number msg1,msg2,msg3,N200,authTag,confTag,empty;
 Function pk
Knowledge :
 alice: alice, mb, inv(pk(alice)), msg1, pk(mb),authTag, confTag;
 bob: bob, mb, msg2, pk(mb),authTag, confTag;
 carsten: carsten, msg3, inv(pk(carsten)), mb, pk(mb), authTag,
     confTag;
 mb: mb, alice, carsten, inv(pk(mb)), pk(alice), pk(carsten),
     authTag, confTag, empty
Actions :
 mb -> alice : N200
 alice -> mb : {authTag,mb, N200,msg1}inv(pk(alice))
 mb -> bob : empty
 bob -> mb : {confTag,bob,mb,msg2}pk(mb)
 mb -> carsten : empty
 carsten -> mb : {confTag,{authTag,mb,msg3}inv(pk(carsten))}pk(mb)
Goals :
 mb authenticates alice on N200,msg1
 msg2 secret between bob,mb
 mb weakly authenticates carsten on msg3
 msg3 secret between carsten,mb
```

Listing 5.5: The automatically generated AnB code for the example with the added information

And, finally, the result of the analysis is shown in Listing 5.6. It tells us that the modeled communication system verifies the specified goals, i.e., that the implementation of the security modules verify their contracts, not just each security modules individually, but also composition of the different exchanged modules.

As with the OFMC analysis result from the Abstract Global level, there are two results being displayed together: the OFMC fixed-point analysis with unbounded number of sessions and the OFMC model-checking analysis of the system with a bounded number of sessions. The GSD framework is currently requesting OFMC to check up to 4 sessions, so that the analysis does not take too long, but that number can be set higher if ones wishes so.

```
Result for default OFMC run:
% Open-Source Fixedpoint Model-Checker version 2012c
Verified for 1 sessions
INPUT
    tmp2069860803.AnB
SUMMARY
  NO_ATTACK_FOUND
GOAL:  honest-weak authentication and secrecy
DETAILS
  UNBOUNDED_NUMBER_OF_SESSIONS, TYPED_MODEL, FREE ALGEBRA
BACKEND  OFMC
COMMENTS  It is recommended to run with --classic option as well (
     see manual).

Result for classic OFMC run:
% Open-Source Fixedpoint Model-Checker version 2012c
INPUT
    tmp2069860803.AnB
SUMMARY
  NO_ATTACK_FOUND
GOAL:  as specified
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
BACKEND  OFMC
STATISTICS
  TIME 405 ms
  parseTime 0 ms
  visitedNodes: 532 nodes
  depth: 13 plies
```

Listing 5.6: OFMC analysis results.

Chapter 6

The Concrete Endpoint Level

The system specification at this level results from splitting the global view of the system into the views of the different principals in the specified system. This makes the system specification closer to code and also to some verification tools, such as LySatool (presented in Section 2.5.1). The global view is split into the views of the different principals using Endpoint Projection, which is described below.

6.1 Endpoint Projection

In order to translate from the Concrete Global level to the Concrete Endpoint level, an endpoint projection is applied. This technique extracts the views of the different principals present in a specification of the global view of the system.

Carbone et al. [CHY07] present the endpoint projection that they use to translate a web-service description of a system in W3C Web Services Choreography Description Language (WS-CDL) [W3C05] — an XML-based language that defines the global behavior of the principals in the system — into applied typed π-calculus.

It is simple to extract the specification of the actions of the sender of the message from the global description of the system because the description of the exchanged message is given from the point of the view of the sender: which elements are being sent and which (if any) encryptions are being performed. For that reason, the result of the endpoint projection for the side of the sender of the message is similar to the specification of the message in the global view of the system.

On the other hand, extracting the specification of the actions of the receiver of the message is a more complex task due to the properties of the modeling language. A modeling language in the style of Alice and Bob lacks a way of specifying the receiver actions and this makes the endpoint projection process harder to perform. This happens because, as mentioned, no information is explicitly modeled in the global view of the system regarding the actions of the receiver of the message, in particular, regarding how the message should be deconstructed, which elements are to be compared with values previously know to the receiver, which values are to be assigned to new variables and which ones are to be decrypted (and the same applies to the information that results from the performed decryptions). Since the structure of the messages specification in the Concrete Global specification is defined in the translation from the Abstract Global level to the Concrete Global level — in the GSD framework prototype with the use of PKI — it is possible to infer the needed information described above in order to derive the required information for specifying the actions of the receiver of a message. Nonetheless, allowing for the possibility of describing receiver actions in the Abstract Global level specification as extra notation could be an interesting line of future research to extend the GSD framework.

The endpoint projection in the GSD framework prototype is performed by going through the system specification at the Concrete Global level and, for each of the actions in the model, generating the correspondent actions performed by the different principals. For example, a message being sent from `A` to `B` in the Concrete Global level is projected to the independent specification of the correspondent actions of `A` and `B`, i.e., `A` would send the message and `B` would receive it, parse it, and decrypt if necessary.

6.2 Concrete Endpoint Level

The syntax for this level is shown in Figure 6.1. It has some similarities with the syntax of the language at the Concrete Global level but the main difference results from the different perspectives of the system in the two levels. The system (*system*) in the Concrete Endpoint level is composed by the sequence of

actions that are performed by the different principals, since the actions at this level do not describe the global view of the system, but the sequence of actions that are performed by each of the principals individually. These actions are either the sending or the receiving and further processing (eventually decrypting, comparing, and assigning elements) of the messages. The latter contain the same elements as the ones in the Concrete Global level, i.e., either plain-text elements or the encryption of elements.

system	::=	**principal: actions** | **system principal: actions**
principal	::=	**string**
actions	::=	**action;** | **actions, action;**
action	::=	**sendMsg** | **receiveMsg** | **createFresh**
createFresh	::=	Create fresh value: **string**
sendMsg	::=	Send message **msg** to **principal**
receiveMsg	::=	Receive message **string** from **principal** (**processMsg**)?
processMsg	::=	**decrypt** (**compare**)? **assign**
decrypt	::=	Decrypt **string** with **key**
compare	::=	Compare beginning of **string** with **string**
assign	::=	Assign remainder of **string** to **string**
msg	::=	**el** | **msg, el**
el	::=	**string** | **encModule**| **hash**
hash	::=	Hash(**msg**)
encModule	::=	**encryptionType**(**msg**, **key**)
encryptionType	::=	shEncryption | pubEncryption
key	::=	PrivKey(**principal**) | PubKey(**principal**) | SharedKey

Figure 6.1: Syntax of the Concrete Endpoint level

6.3 The Message Board

The model of our message board example at this level is shown in Listing 6.1.

```
------------------ Principal Alice ------------------
Receive message N200 from MB;
Send message pubEncryption("authTag, MB, N200, msg1", PrivKey(Alice
    )) to MB;

------------------ Principal Bob ------------------
Send message pubEncryption("confTag, msg2", PubKey(MB)) to MB;

------------------ Principal Carsten ------------------
Send message pubEncryption("confTag, pubEncryption("authTag, MB,
    msg3", PrivKey(Carsten))", PubKey(MB)) to MB;

------------------ Principal MB ------------------
Create fresh value: N200;
Send message N200 to Alice;
Receive message x833 from Alice;
Decrypt x833 with PubKey(Alice)
Compare beginning of x833 to "authTag,MB,N200"
Assign remainder of x833 to msg1


Receive message x863 from Bob;
Decrypt x863 with PrivKey(MB)
Compare beginning of x863 to "confTag"
Assign remainder of x863 to msg2


Receive message x454 from Carsten;
Decrypt x454 with PrivKey(MB)
Compare beginning of x454 to "confTag"
Assign remainder of x454 to x190

Decrypt x190 with PubKey(Carsten)
Compare beginning of x190 to "authTag,MB"
Assign remainder of x190 to msg3
```

Listing 6.1: The messages from the system in the Concrete Endpoint level.

6.4 Concrete Endpoint Level Outputs

At this level, the GSD framework connects with LySatool and it also outputs the code implementation of the modeled system.

6.4.1 The LySatool Analysis

As presented in Section 2.5.1, LySatool [Buc05] performs security analysis of protocols described in LySa [BBD+05]. The LySa code that is automatically generated by the GSD framework prototype is shown in Listing 6.2.

```
1  (new +- KAlice)
2  (Alice,MB;N200).
3  <Alice,MB,{|authTag,MB,N200,msg1|}:KAlice->.
4  0
5  |
6  (new +- KBob)
7  <Bob,MB,{|confTag,msg2|}:KMB+>.
8  0
9  |
10 (new +- KCarsten)
11 <Carsten,MB,{|confTag,{|authTag,MB,msg3|}:KCarsten-|}:KMB+>.
12 0
13 |
14 (new +- KMB)
15 (new N200)
16 <MB,Alice,N200>.
17 (MB,Alice;x339).
18 decrypt x339 as {|authTag,MB,N200;msg1|}:KAlice+ in
19 (MB,Bob;x253).
20 decrypt x253 as {|confTag;msg2|}:KMB- in
21 (MB,Carsten;x513).
22 decrypt x513 as {|confTag;x439|}:KMB- in
23 decrypt x439 as {|authTag,MB;msg3|}:KCarsten+ in
24 0
```

Listing 6.2: The automatically generated LySa code of the example system.

Having the automatically output LySa code, there are two different pieces of information that have to be added manually before the LySatool can be executed. Those are the so called cryptopoints and also the declaration of freshly generated elements. The completed code is shown in Listing 6.3. The cryptopoints are modeled inside square brackets (first seen in the end of line 4) and provide the necessary information to LySatool regarding where the encrypted elements are supposed to be decrypted and also regarding where the decrypted elements should have been encrypted originally. For example, the cryptopoint in line 4 models the fact that the encryption can either be decrypted at destination `b` (which can be seen in line 21) or at destination `CPDY`, which models the attacker. The cryptopoint models that the message can be decrypted by the attacker because it is in fact a message signed by `Alice` and can therefore be decrypted using her public-key, which the attacker has access to. Another example is the cryptopoint in line 23, which models that the encryption of the decrypted message should have been done either at cryptopoint `c` (in line 9) or at cryptopoint `CPDY`. The message could have been encrypted by the attacker

because it is a message encrypted with MB's public-key, which the attacker also has access to.

The other added piece of information is the declaration of freshly generated elements. The first one can be seen in line 3. These declarations have to be specified because otherwise the messages would be interpreted by LySatool as being known by everyone, including the attacker, which would invalidate the confidentiality analysis of the system.

These two pieces of information can, in fact, be extracted automatically from the system model present in the GSD framework and that is part of future work, as mentioned in Section 9.1.

```
1  (new +- KAlice)
2  (Alice,MB;N200).
3  (new msg1)
4  <Alice,MB,{|authTag,MB,N200,msg1|}:KAlice- [at a dest {b,CPDY}]>.
5  0
6  |
7  (new +- KBob)
8  (new msg2)
9  <Bob,MB,{|confTag,msg2|}:KMB+ [at c dest {d}] >.
10 0
11 |
12 (new +- KCarsten)
13 (new msg3)
14 <Carsten,MB,{|confTag,{|authTag,MB,msg3|}:KCarsten- [at g dest {h,
       CPDY}] |}:KMB+ [at e dest {f}]>.
15 0
16 |
17 (new +- KMB)
18 (new N200)
19 <MB,Alice,N200>.
20 (MB,Alice;x3).
21 decrypt x3 as {|authTag,MB,N200;msg1|}:KAlice+ [at b orig {a}] in
22 (MB,Bob;x313).
23 decrypt x313 as {|confTag;msg2|}:KMB- [at d orig {c,CPDY}] in
24 (MB,Carsten;x468).
25 decrypt x468 as {|confTag;x547|}:KMB- [at f orig {e,CPDY}] in
26 decrypt x547 as {|authTag,MB;msg3|}:KCarsten+ [at h orig {g}] in
27 0
```

Listing 6.3: The completed LySa code of the example system.

Using the code from Listing 6.2 as input to LySatool, the obtained result is shown in Figure 6.2. There is a big amount of information in this result from LySatool, but the most important are the two first fields: *Values that may not be confidential* and *Violation of authentication properties*. The former, as the name indicates, displays the set of data that might not be confidential after the protocol run and, in this case, it is important to see that neither `msg2` nor `msg3`

are in that set, which is the intention since they are transmitted, respectively, in a confidential and secure way.

LySatool result: Values that may not be confidential
N200, $n_\bullet$, $m_\bullet$+, $m_\bullet$-, KMB+, Alice, KAlice+, Bob, Carsten, confTag, KCarsten+, authTag, MB, $\{|$LauthTag, LMB, $l_\bullet$, Lmsg1$|\}_{\text{LKAlice-}}$ [at a dest { b, CPDY }], $\{|$LconfTag, L33$|\}_{\text{LKMB+}}$ [at e dest { f }], $\{|$LconfTag, Lmsg2$|\}_{\text{LKMB+}}$ [at c dest { d }], $\{|l_\bullet, l_\bullet, l_\bullet|\}_{l_\bullet}$ [at CPDY], $\{|l_\bullet, l_\bullet|\}_{l_\bullet}$ [at CPDY], $\{|l_\bullet, l_\bullet, l_\bullet, l_\bullet|\}_{l_\bullet}$ [at CPDY], msg1

Violation of authentication properties (ψ)
(e, d), (c, f)

Variable bindings (ϱ)
ϱ($x_\bullet$) = LN200, $l_\bullet$, L13, LAlice, LMB, LCarsten, L36, LBob, L22, LauthTag, Lmsg1

ϱ(msg1) = Lmsg1

ϱ(msg2) = L33, Lmsg2, $l_\bullet$

ϱ(msg3) = Lmsg3

ϱ(x547) = L33, Lmsg2, $l_\bullet$

Messages communicated (ϰ)
<LAlice, LMB, L13>, <$l_\bullet$, $l_\bullet$, $l_\bullet$>, <LMB, LAlice, LN200>, <LCarsten, LMB, L36>, <LBob, LMB, L22>, <$l_\bullet$>

Tree grammars (γ)
$l_\bullet$ → N200, $n_\bullet$, $m_\bullet$+, $m_\bullet$-, KMB+, Alice, KAlice+, Bob, Carsten, confTag, KCarsten+, authTag, MB, $\{|$LauthTag, LMB, $l_\bullet$, Lmsg1$|\}_{\text{LKAlice-}}$ [at a dest { b, CPDY }], $\{|$LconfTag, L33$|\}_{\text{LKMB+}}$ [at e dest { f }], $\{|$LconfTag, Lmsg2$|\}_{\text{LKMB+}}$ [at c dest { d }], $\{|l_\bullet, l_\bullet, l_\bullet|\}_{l_\bullet}$ [at CPDY], $\{|l_\bullet, l_\bullet|\}_{l_\bullet}$ [at CPDY], $\{|l_\bullet, l_\bullet, l_\bullet, l_\bullet|\}_{l_\bullet}$ [at CPDY], msg1

LKMB+ → KMB+

LKAlice+ → KAlice+

LKMB- → KMB-

LKCarsten+ → KCarsten+

LKAlice- → KAlice-

LKCarsten- → KCarsten-

L13 → $\{|$LauthTag, LMB, $l_\bullet$, Lmsg1$|\}_{\text{LKAlice-}}$ [at a dest { b, CPDY }]

L22 → $\{|$LconfTag, Lmsg2$|\}_{\text{LKMB+}}$ [at c dest { d }]

L33 → $\{|$LauthTag, LMB, Lmsg3$|\}_{\text{LKCarsten-}}$ [at g dest { h, CPDY }]

L36 → $\{|$LconfTag, L33$|\}_{\text{LKMB+}}$ [at e dest { f }]

LMB → MB

LBob → Bob

LN200 → N200

Lmsg1 → msg1

Lmsg2 → msg2

Lmsg3 → msg3

Lx547 → $\{|$LauthTag, LMB, Lmsg3$|\}_{\text{LKCarsten-}}$ [at g dest { h, CPDY }], msg2, msg1, $\{|l_\bullet, l_\bullet, l_\bullet|\}_{l_\bullet}$ [at

Figure 6.2: Output from LySatool

The other important field, *Violation of authentication properties*, tells us that there are two violations possibles, denoted by `(e,d)` and `(c,f)`. Regarding `(e,d)`, by looking at the code being analyzed by LySatool (Listing 6.3), one can see that cryptopoint `e` corresponds to the message being sent on line 14 (of Listing 6.2) by `Carsten` can be received and decrypted by `MB` on line 23, which is modeling the message that `MB` received from `Bob`. This result denotes a replay attack where `MB` would accept and decrypt the message from `Carsten` as if it was from `Bob`. Although, it is worth noting that the result of that decryption would be an encrypted element and not a humanly-readable message, as `MB` was expecting. As for the other violation, `(c,f)` represents an attack where `MB` would receive and decrypt a message (in line 25) from `Bob` as if it was from `Carsten`. The main difference from this violation to the previous one is that `MB` further decrypts the message (in line 26) and would notice that an attack was being done there, since the decryption would not work. This happens because the first violation is related with the reception of a confidential message while the second violation is related with the reception of an secure message. Overall, this is an interesting result that results from the compositionality of security modules and it highlights why it is still useful to perform analysis of the security properties after implementing the security modules.

6.4.2 The Code Output

The GSD framework outputs the system model to code by replacing the different elements of the Concrete Endpoint specification with pre-determined Java blocks of code that implement those elements. Furthermore, each of those blocks also makes use of a Java library for functions such as encryption, decryption, and generation of nonces.

The translation outputs Java code for each of the modeled principals individually. In Listing 6.4, the Java code for Alice in the message board running example is shown. The code for the Message Board (MB) is shown in Listing 6.5. The automatically generated code is the skeleton for the communication part of system. This skeleton can then be used in the programs of each of the principals to perform the communications. The code uses functions from a developed Java library, adapted from my Master Thesis work [Qua10], which contains functions for encryption and decryption of elements, generation of unique values, and connection establishment. The code for this library is shown in Appendix A.1. The output code for `Bob` and `Carsten` is not shown here since, except for the absence of the nonce and the different encrypted elements, it is similar to the output code of Alice.

Part of future work for the further development of the GSD framework tool prototype is to generate not just the communication skeleton but java programs that can be compiled and executed.

```
/*START RECEIVE MSG*/
if(!(sockMB.isBound())) {
    sockMB = makeConnection(Alice, MB, myPortMB, addressOfMB,
        portOfMB);
    outMB = new PrintWriter(sockMB.getOutputStream(), true);
    inMB = new BufferedReader(new InputStreamReader(sockMB.
        getInputStream()));
}

String N200 = "";
inputLine = inMB.readLine();
N200=inputLine+'\n';

System.out.println("Received msg from MB: " +N200);
/*FINISH RECEIVE MSG*/

/*START SEND MSG*/
if(!(sockMB.isBound())) {    sockMB = makeConnection(Alice, MB,
    myPortMB, addressOfMB, portOfMB);
    outMB = new PrintWriter(sockMB.getOutputStream(), true);
    inMB = new BufferedReader(new InputStreamReader(sockMB.
        getInputStream()));
}

System.out.println("Sending msg to MB: pubEncryption("authTag,MB," +
    N200 + "," + msg1, PrivKey(Alice))");
outMB.println(pubEncryption("authTag,MB," + N200 + "," + msg1,
    PrivKey(Alice)));
/*FINISH SEND MSG*/
```

Listing 6.4: Java code for Alice in the message board example.

```
String N200 = generateNonce();

/*START SEND MSG*/
if(!(sockAlice.isBound())) {    sockAlice = makeConnection(MB, Alice
    , myPortAlice, addressOfAlice, portOfAlice);
   outAlice = new PrintWriter(sockAlice.getOutputStream(), true);
   inAlice = new BufferedReader(new InputStreamReader(sockAlice.
       getInputStream()));
}

System.out.println("Sending msg to Alice: N200");
outAlice.println(N200);
/*FINISH SEND MSG*/

/*START RECEIVE MSG*/
if(!(sockAlice.isBound())) {
   sockAlice = makeConnection(MB, Alice, myPortAlice,
       addressOfAlice, portOfAlice);
   outAlice = new PrintWriter(sockAlice.getOutputStream(), true);
   inAlice = new BufferedReader(new InputStreamReader(sockAlice.
       getInputStream()));
}

String x133 = "";
inputLine = inAlice.readLine();
x133=inputLine+'\n';

System.out.println("Received msg from Alice: " +x133);
/*FINISH RECEIVE MSG*/

/*Start of message decryptions,comparisons,and assigns*/
String decx133 = pubDecrypt(x133,PubKey(Alice));
if !(decx133.startsWith("authTag,MB,N200")) Die("Unmatched elements
    after decryption:" + x1.substring(0,x2.length) + " and " + x2)
    ;
String msg1 = decx133.substring("authTag,MB,N200".length()+1);

/*End of message decryptions,comparisons,and assigns*/
/*START RECEIVE MSG*/
if(!(sockBob.isBound())) {
   sockBob = makeConnection(MB, Bob, myPortBob, addressOfBob,
       portOfBob);
   outBob = new PrintWriter(sockBob.getOutputStream(), true);
   inBob = new BufferedReader(new InputStreamReader(sockBob.
       getInputStream()));
}

String x473 = "";
inputLine = inBob.readLine();
x473=inputLine+'\n';

System.out.println("Received msg from Bob: " +x473);
/*FINISH RECEIVE MSG*/

```

```
47 /*Start of message decryptions,comparisons,and assigns*/
48 String decx473 = pubDecrypt(x473,PrivKey(MB));
49 if !(decx473.startsWith("confTag")) Die("Unmatched elements after
       decryption:" + x1.substring(0,x2.length) + " and " + x2);
50 String msg2 = decx473.substring("confTag".length()+1);
51
52 /*End of message decryptions,comparisons,and assigns*/
53 /*START RECEIVE MSG*/
54 if(!(sockCarsten.isBound())) {
55    sockCarsten = makeConnection(MB, Carsten, myPortCarsten,
          addressOfCarsten, portOfCarsten);
56    outCarsten = new PrintWriter(sockCarsten.getOutputStream(), true
          );
57    inCarsten = new BufferedReader(new InputStreamReader(sockCarsten
          .getInputStream()));
58 }
59
60 String x239 = "";
61 inputLine = inCarsten.readLine();
62 x239=inputLine+'\n';
63
64 System.out.println("Received msg from Carsten: " +x239);
65 /*FINISH RECEIVE MSG*/
66
67 /*Start of message decryptions,comparisons,and assigns*/
68 String decx239 = pubDecrypt(x239,PrivKey(MB));
69 if !(decx239.startsWith("confTag")) Die("Unmatched elements after
       decryption:" + x1.substring(0,x2.length) + " and " + x2);
70 String x900 = decx239.substring("confTag".length()+1);
71
72 String decx900 = pubDecrypt(x900,PubKey(Carsten));
73 if !(decx900.startsWith("authTag,MB")) Die("Unmatched elements
       after decryption:" + x1.substring(0,x2.length) + " and " + x2);
74 String msg3 = decx900.substring("authTag,MB".length()+1);
75
76 /*End of message decryptions,comparisons,and assigns*/
```

Listing 6.5: Java code for the Message Board (MB) in the message board example.

Chapter 7

Tool Implementation

In this chapter, specific details regarding the implementation of the GSD framework are given. Initially, the motivations behind the implementation choices are presented in Section 7.1. In Section 7.2, the implementation of the GSD framework is introduced and, finally, the implementation of the BAK tool is presented in Section 7.3.

7.1 Framework choices

After the investigation of the state-of-the-art on the different components of the framework presented in Chapter 2, a solution that enabled to reach the goals of the presented research was found.

As mentioned before, the solution that was reached was based on a language similar to the Alice and Bob notation, similar throughout the whole framework but with some differences between the different abstraction levels. Alice and Bob notation is a simple and intuitive language when modeling communication systems and it also makes it simple to automatically translate to AnB, which is used by the OFMC tool and it makes it also possible to translate to the LySa language and then be analyzed by LySatool. It is also possible to translate that

input language into assertions that can be used with the Beliefs and Logic tools.

7.2 Implementation of the GSD Framework Prototype

Having in consideration what the framework consists of, mostly parsing of the input code, transformation of the code, which entails the interpretation and traversal of abstract syntax trees (AST), I opted for using a functional language for the implementation of the framework. That functional language was F# and the prototype was developed in Visual Studio. The prototype consists of a GUI front-end (shown in Figure 7.1) and the code that supports the GUI and all the framework's functionalities.

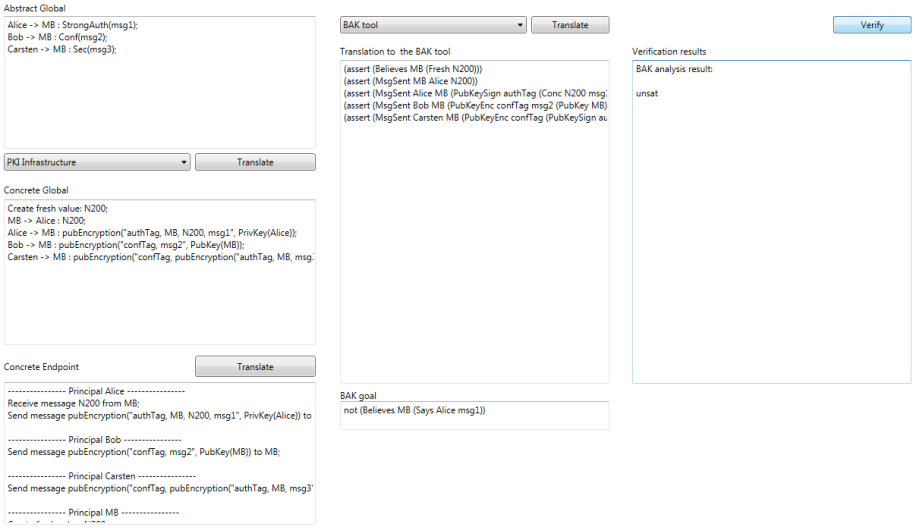


Figure 7.1: The front-end of the GSD framework prototype.

The overall functioning of the framework is inspired by the method of using ANTLR that was presented in Section 2.8.1 and can be seen in Figure 7.2. There are several tree walkers in the tool with different functionalities: process the abstract syntax tree, translate the system specification between the different abstraction levels, and output code.

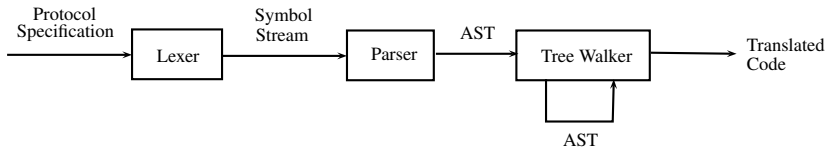


Figure 7.2: Overall GSD tool process

7.2.1 Implementation of the Security Modules Contracts in the AnB Code

In this Section, the implementation of the contracts of the security modules for OFMC is presented. The contracts are used when generating the AnB code that is verified by OFMC to generate the security goals of the communication system. The contracts can therefore be implemented as a conjugation of AnB goals as follows:

Authentication. In order to have the sent message verified for authentication, the AnB goal states that the receiver authenticates another principal via that received message. In fact, for the authentication module, `weakly authenticates` should be used, since that goal does not require any freshness properties. Therefore, the goal for an authenticated message `msg` send from `X` to `Y` is:

Auth: `Y weakly authenticates X on msg`

Strong Authentication This contract is similar to the authentication contract, but with one important difference: it requires freshness assurances, avoiding any replay attacks. That is reflected in the contract implementation shown bellow, where `X` sent a strongly authenticated message to `Y`:

Strong Auth: `Y authenticates X on msg`

Confidentiality. In order to achieve confidentiality, the exchanged message should be a secret between the sender and the intended receiver. In AnB syntax, the implemented contract for a message `msg` sent confidentially from `X` to`Y` that is modeled by having:

Conf: `msg secret between X,Y`

Security. As for security, it can be seen as the composition of authentication and confidentiality. Therefore, the contract implementation in AnB for a secure message `msg` sent from `X` to`Y` has the following two goals:

Sec: `Y weakly authenticates X on msg , msg secret between X,Y`

Strong Security. This contract is similar to the one presented above, but it is the composition of the strong authentication and the confidentiality contracts. The contract implementation in AnB for a strongly secure message `msg` sent from `X` to`Y` has the following two goals:

StrongSec: `Y authenticates X on msg , msg secret between X,Y`

7.3 Implementation of the BAK Tool

In this Section, specific details regarding the implementation of the BAK tool are presented.

7.3.1 Implementation of the Security Modules Contracts in the BAK Tool

In this Section, the implementation of the security modules contracts for the BAK tool is given. The elements of these contracts are used as assertions together with the system model and the core inference rules to determine whether the contracts hold for the chosen implementation of the security modules.

Authentication. In order to have the sent message authenticated, the receiver of the message must believe that it was the sender who actually sent it. For example, if X sends an authenticated message to Y, the desired outcome is the following:

Auth: `Believes(Y,Said(X,m))`

Strong Authentication This contract is similar to the authentication contract, but with one important difference: it not only requires that the receiver believes that the sender sent the message, but also that the message was recently sent. That is the difference, in the used logic, between `Said` and `Says`. That is reflected in the contract shown bellow, where X sent a strongly authenticated message to Y:

Strong Auth: `Believes(Y,Says(X,m))`

Confidentiality. In order to achieve confidentiality, the exchanged message can only be seen by the rightful receiver and not seen by an attacker that might have access to the communication. Therefore, the contract for a confidential message sent from X to Y is the following:

Conf: `Sees(Y,m)` $\wedge$ $\neg$`(Sees(attacker,m))`

Security. As for security, it can be seen as the composition of authentication and confidentiality. The receiver of the message should be entitled to believe that the message was sent by the original sender and the receiver should also be the only one being able to see the message itself. In fact, the beliefs that are required for the security contract are the union of the required beliefs for the authentication and confidentiality contracts. Therefore, the contract for a secure message sent from X to Y is the following:

Sec: `Believes(Y,Said(X,m))` $\wedge$ `Sees(Y,m)` $\wedge$ $\neg$`(Sees(attacker,m))`

Strong Security. This contract is similar to the one presented above, but it is the composition of the strong authentication and the confidentiality contracts, therefore adding freshness to the security contract is, once again, represented by the difference between `Said` and `Says`. The contract for a strongly secure message from X to Y is the following:

Strong Sec: `Believes(Y,Says(X,m))` $\wedge$ `Sees(Y,m)` $\wedge$
$\neg$`(Sees(attacker,m))`

7.3.2 Implementing the Inference Rules

As an example of the implementation of the inference rules of the BAK tool engine, we revisit one of the rules presented Section 3.1.2.2:

$$Sees(X, Sign(el, PrivKey(Y))) \wedge Sees(X, PubKey(Y))$$
$$\Longrightarrow Sees(X, el) \wedge Believes(X, Said(Y, el))$$

The rule, in the SMT-LIB2 language, is shown in Listing 7.1. It specifies which beliefs a principal can infer from a message signed with a private-key: if a principal sees an element signed with a private-key and if he knows the correspondent public-key, then he is able to decrypt it, see the element that had been signed, and have assurances on which principal signed the element.

Furthermore, lines 7 is also important for the implementation of the tool since it creates a pattern that helps Z3 instantiating the variable named in line 2. Furthermore, the name specified in the end of line 7 is used when returning the unsatisfiability core. Naming this rule allows the tool to present the name when returning an unsatisfiability core that contains the rule.

```
1 ;PRIVATE DECRYPTION
2 (assert (! (forall ((x Principal) (w Principal) (el Element) (tag
      Element))
3       (! (=> (and (Sees x (PubKeySign tag el (PrivKey w)))
4                   (Sees x (PubKey w)))
5              (and (Sees x el)
6                   (Believes x (Said w el))))
7         :pattern ((Sees x (PubKeySign tag el (PrivKey w))) (Sees x (
              PubKey w))))) :named decOfPrivKeySign))
```

Listing 7.1: One of the system rules regarding decryption.

7.3.3 Implementing the Domain restriction

Domain restriction is implemented by restricting the possible number of distinct elements in the system. The number of elements is a potential problem because there are functions in the system, such as concatenation and encryption, that generate new elements from existing elements and, if this generation of new elements is not controlled, then the system will be infinitely generating new elements, which would lead to non-termination.

The approach taken was to implement a specific number of domain levels and by placing a newly generated element in a domain level higher than the highest domain level of the elements used to create that new element. This can be seen, for example, in the concatenation rule shown in Listing 7.2. More specifically, lines 4 and 5 model the fact that only elements from the first or second domain are allowed to be concatenated to create another element, therefore restricting the (otherwise infinite) concatenation of elements.

```
1 (assert (forall ((el1 Element) (el2 Element))
2     (! (=> (and (Sees attacker el1)
3                 (Sees attacker el2)
4                 (or (firstDomain el1) (secondDomain el1))
5                 (or (firstDomain el2) (secondDomain el2)))
6            (Sees attacker (Conc el1 el2)))
7        :pattern ((Sees attacker el1) (Sees attacker el2)))))
```

Listing 7.2: Rule for concatenation exemplifying the domain restriction

The other part of the implementation of this restriction is shown in Listing 7.3, where the two depicted rules implement the assignment of the domain level of the element generated by the concatenation. The domain restriction process is similar in all the other rules present in the system that generate new elements.

```
1 (assert (forall ((el1 Element) (el2 Element))
2   (=> (and (firstDomain el1) (firstDomain el2))
3       (secondDomain (Conc el1 el2)))))
4
5 (assert (forall ((el1 Element) (el2 Element))
6   (=> (and
7           (or (firstDomain el1) (secondDomain el2))
8           (or (secondDomain el1) (firstDomain el2))
9           (or (secondDomain el1) (secondDomain el2)))
10      (thirdDomain (Conc el1 el2)))))
```

Listing 7.3: Rule for concatenation exemplifying the domain restriction

In summary, the domain in the BAK tool analysis is restricted in two ways. By restricting the number of elements in each of the domain levels and by restricting the number of domain levels. Both restrictions can be easily changed in the implementations, so that one is sure that there are enough number of elements and enough number of levels to get correct analysis results. For example, there are currently three domain levels in the prototype implementation and, for that reason, the system cannot reason about more than three nested concatenations. If a system model has more than three nested concatenations, then the necessary number of levels should be add to the system.

7.3.4 Modeling the Represents Rule

In Section 4.2.2, the semantics of the security modules is presented and one of the used functions introduced there is *represents*. This can be used, for example, as X *represents* Y. This function models the fact that X has the knowledge required to represent Y, i.e., to act as Y. A basic rule in our system is that X represents itself and it might be able to represent another principal in case X acquires the necessary knowledge for that.

In our prototype implementation, and since we are using a Public-Key Infrastructure, the *represents* function is modeled directly by the knowledge of a principal's private-key. This is not the best solution to implement delegation, however, the goal of this prototype implementation is to demonstrate how the framework can be implemented and, therefore, the *represents* rule is implemented in this simplified as an illustrative example. Having this in consideration, the implementation of, for example, the rule for the confidentiality module shown in Section 4.2.2.3 is the following:

$$\frac{Z \ sees \ Conf(X, w), \quad Z \ represents \ X}{Z \ sees \ w}$$

is given by the rule presented in Section 3.1.2.2:

$$\frac{Z \ sees \ Enc(w, PubKey(X)), \ Z \ sees \ PrivKey(X)}{Z \ sees \ w}$$

where the confidentiality module is implemented with the encryption with the public-key of Y and Z *represents* X is modeled by the fact that Z sees X's private-key.

Chapter 8

Use Case: Verifying ADS-B Communication System

The main goal of this use case is to model and analyze the ADS-B airspace navigation system [QR]. The research performed in this chapter was performed in my External Stay at NASA Ames Research Center in collaboration with Kristin Y. Rozier.

ADS-B stands for Automatic Dependent-Surveillance Broadcast and is a new airspace navigation system being deployed with the goal of complementing (and eventually subsume) the current airspace navigation system (i.e. the radar). Its main advantage in relation to the radar is its ability to provide a more accurate aircraft location information. ADS-B will, therefore, play a crucial role in world-wide aviation and, in particular, international commercial aviation. The system is presented in more detail in Section 8.1.

Due to the crucial role it is going to play, we believe it is natural to study its security properties. In fact, we are not the first ones to believe that: McCallie identified different ADS-B attack taxonomies [MBM11] that we discuss and model in Section 8.3.2.1 and Valovage et al. made several contributions on extending ADS-B by suggesting different ways of making it more secure [Val06, VV+07]. His suggestions did have some issues, and part of the work presented here aims at solving those issues and at providing suggestions for further improvement of

the system. Valovage's extensions are presented and discussed in Section 8.3.3 and our suggestions are presented in Section 8.4.

We used the GSD framework to model and verify the current ADS-B communication system. Having the formal results of the framework showing us how insecure the ADS-B communication system is by itself gave us more confidence for the modeling and analysis of the different attack scenarios and those eventually lead us to investigate and successfully verify possible security extensions of ADS-B.

It is worth mentioning that the scope of this analysis is limited to ADS-B and not to the airspace system as a whole. ADS-B is a new airspace navigation system, but it is a part of a much larger system that is used to control the airspace. Therefore, the analysis results presented here do not reflect the security of the whole airspace system but of ADS-B by itself.

8.1 ADS-B

As mentioned above, ADS-B [FAAa, RTC02] is an air traffic management surveillance system that is a crucial part of NextGen [FAAb]. As depicted in Figure 8.1, ADS-B is a large wireless network, composed by ground stations and aircraft that communicate with each other — the aircraft report flight information (such as their position, velocity, and intent) and receive traffic and other information from the ground stations — in a way that enhances the "see and avoid" capability of the aircraft. In fact, the main benefit of ADS-B is the provided higher accuracy regarding the aircraft position, which is crucial in an airspace that is increasingly more crowded with aircraft.

Figure 8.1: Overview of the ADS-B system

ADS-B has two main components: ADS-B Out and ADS-B In. ADS-B Out — the broadcast of the aircraft's information — will be mandatory in the American National Airspace System (NAS) by 2020. ADS-B In — the reception of the information sent by the base stations and also the reception of ADS-B Out broadcasted by other aircraft — is currently being studied and it is still not clear when, or even if, it will be mandatory.

There is no doubt that ADS-B will have a crucial role in the aircraft navigation in the NAS and, for that reason, it is essential that it is formally verified before its final adoption. One potential vulnerability of the system is the lack of built-in security in its communications. In fact, it currently relies on current systems (such as the radar) to achieve the security guarantees. We believe that this approach is not the ideal design approach and that it is potentially dangerous to deploy a new system that has to rely on current systems to fully function. Therefore, we believe that it is important to verify the security properties of ADS-B by itself, i.e.; who is able to send and receive the different exchanged messages. For example, are the exchanged messages authenticated? I.e., is an attacker able to send information to an aircraft, impersonating a base station? Are the messages confidential? I.e., as an attacker able to read the information broadcasted by the aircraft and therefore have access to the exact location of the aircraft. The argument is not whether (or when) authentication and/or confidentiality are in fact needed in ADS-B. The goal of this use case is to interpret the results of the formal analysis of the different security aspects of ADS-B and its suggested extensions.

8.1.1 ADS-B's Communication System

The legitimate principals taking part in the ADS-B's communication system are the aircraft and the ground-station. As previously mentioned, ADS-B has two components that allow the principals to communicate: ADS-B Out and ADS-B In.

ADS-B Out consists of the messages that are broadcasted by the aircraft. A message contains the aircraft's position and speed (both acquired through a positioning system, presently GPS) and potentially other information, such as intent. These broadcasted messages are received by the ground-stations and, in case ADS-B In is being used, the former will also be received by the aircraft that are within range of the broadcaster aircraft.

ADS-B In concerns the capability of receiving the ADS-B Out messages. As explained above, an aircraft is only able to receive air to air ADS-B messages broadcasted from other aircraft if it has ADS-B In capabilities. Another part of

ADS-B In is the information broadcasted by the ground-stations. This consists of traffic and weather information. During the transitional phase, the traffic information will have a mixture of ADS-B and Radar information, enabling the ADS-B In equipped aircraft to have a full view of the airspace surrounding it.

In the U.S., ADS-B can be used in two different data-links: the 1090MHz Extended Squitter and the Universal Access Transceiver (UAT).

8.1.1.1 1090MHz Extended Squitter

ADS-B can be used in this medium by using the Mode S transponder with Extended (112 bit) Squitter and aircraft flying above 18000 feet are obliged to used this medium. It is worth noting that this is the only allowed technology for ADS-B communication in the rest of the world.

An ADS-B message in this format has 112bits of which 8 bits are reserved for control codes (such as message type and sub-type), 24 bits for the aircraft (ICAO) address, 56 bits for ADS-B data and, finally, 24 bits for Forward Error Correction (FEC), which is able to detect and correct a limited number of errors without the need to re-transmitting the data.

Position and Velocity data is sent in different messages based on the given type codes [RTC11a, table 2-14]. For an airborne position message, altitude is 12 bits in either 100 or 25 foot increments, depending on the chosen setting. Longitude and latitude are 17 bits each in Compact Position Reporting (CPR) encoded format [RTC11a, appendix T]. This format was specifically developed for ADS-B messages broadcast on this data-link with the goal of reducing the number of bits required in the transmission of the longitude and latitude coordinates. The velocity message is a map between integer range from 1 to 1023 and a range of speeds in knots depending on whether the aircraft is traveling on the ground, subsonic, or supersonic. It can be either in E-W and N-S vector components plus vertical rate or in a Heading, Airspeed, and Vertical Rate format [RTC11a, tables A-1, A-5, A-6].

8.1.1.2 Universal Access Transceiver

This data-link (978MHz) can only be used for ADS-B broadcast in the NAS and by small, non-commercial aircraft that fly bellow 18000 feet. The American Federal Aviation Administration (FAA) decided to also allow ADS-B to be used in this medium because it makes it not only cheaper but also easier to start

using ADS-B since no new hardware has to be installed in most of the small aircraft flying today. This decision will, however, also result in a more complex system, since aircraft using different data-link technologies will not be able to communication directly using ADS-B and will only receive each other's ADS-B messages via the ADS-R broadcasts (see Section 8.1.1.3).

There are two basic types of broadcast messages on the UAT channel: the ADS-B Message, and the Ground Uplink Message [RTC11b, page 24]. The ADS-B Message is broadcasted by an aircraft to share its position, velocity and other information while the Ground Uplink Message is used by ground stations to up-link flight information to any aircraft that is within its reach. An ADS-B Message in the UAT link has two main components, one that contains user information and another used for forward error correction code parity, that supports the correction of a limited number of errors on the transfer of the data. To achieve this FEC code parity, Reed-Solomon blocks [RS60] are used in the UAT link. This technique uses finite field arithmetic and its widespread use goes from the CD player to communication systems, including spacecraft communications [Wic94].

The aforementioned messages contain a payload/application data of 144 bits (Basic ADS-B Message) or 272 bits (Long ADS-B Message) [RTC11b, page 24]. The Ground Uplink Message can have up to 3455bits of payload data, of which 64bits are reserved for a UAT Specific Header and the remaining 3392bits can be used for application data [RTC11b, page 26]. [1]

8.1.1.3 ADS-R

ADS-R stands for Automatic Dependent Surveillance - Rebroadcast and its purpose is to translate the ADS-B messages from one technology to the other. That is, if an ADS-R station receives an ADS-B broadcast in the 1090MHz frequency, it will reformat it and broadcast it in the 978MHz frequency (UAT) and vice-versa. This service was created to mitigate the incompatibility of the ADS-B messages being broadcasted in the two different frequencies.

8.1.1.4 ADS-C

Automatic Dependent Surveillance Contract (ADS-C, but sometimes also called ADS-A) is identical to ADS-B, but as opposed to ADS-B, the information

[1] The number of bits available for the Ground Uplink Message differs from that in Valovage's work [Val06]; it is from [RTC11b], which was published 5 years later.

reaches the ground stations using Satellite communication, i.e., the Mode S Transponder transmits the information via a Satcom, which is then relayed to the ground stations. ADS-C could, therefore, be the main method of communication in areas outside the range of the any ground stations, such as large bodies of water [Avi12].

8.2 Using the GSD framework

The usage of GSD framework for modeling and analyzing ADS-B and its potential extensions is shown in Figure 8.2. The most abstract level of the framework (Abstract Global) was not used, since that level is more targeted for developing secure systems and not so much for modeling and analyzing systems previously developed. This happens because the Abstract Global level's strength is the usage of the different security modules, which are useful when developing a system from scratch and we want to express the desired security properties of the data in the messages. The Concrete Endpoint level was not used either, since we were not focusing on the analysis and code output performed at that level. The analysis was performed using the BAK tool 3 since it is the most flexible tool of the ones used by the GSD framework and we were, therefore, able to tailor that analysis to the system being analyzed.

For these reasons, we used the ADS-B system model as input to the Concrete Global level. This model is automatically translated into a language that can be used by the BAK tool to verify the system and its properties (or goals) as detailed in Section 3.2.

One necessary change in the inference rules of the BAK tool had to be performed regarding the way the sending of messages is processed. In ADS-B, a principal broadcasts the messages instead of sending them directly to another principal and that is reflected in the adapted model of the system, where the messages are specified as being sent to a specific principal (*Air*) in order to model their broadcast. Therefore, the rule was changed to:

$$\textbf{MsgSent(X,Air,el)} \implies \textbf{Sees(P,el)}$$

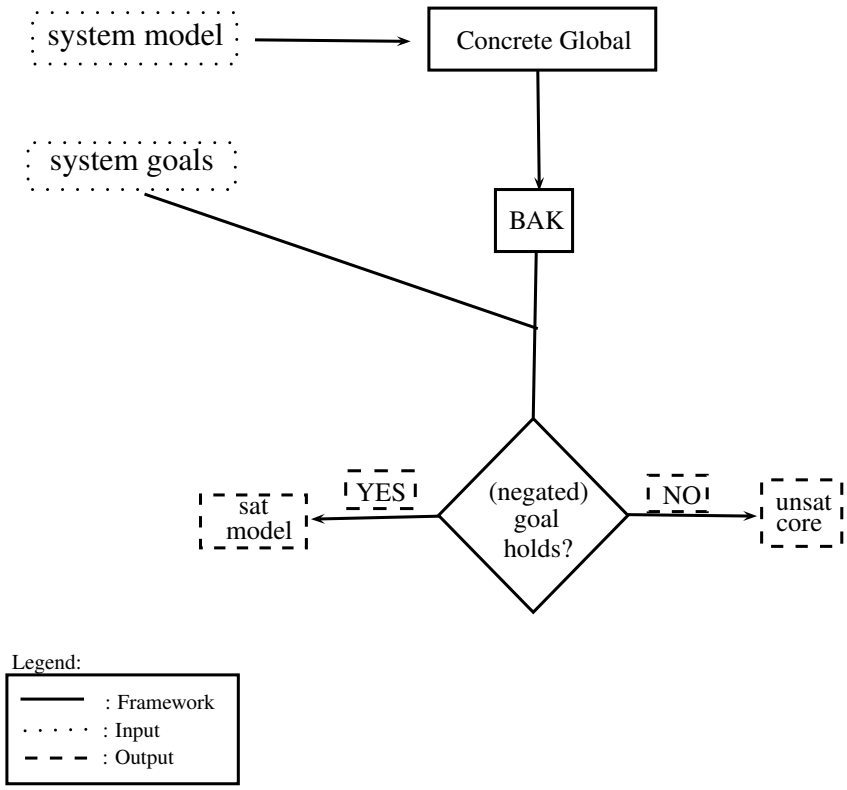


Figure 8.2: The GSD Framework applied to ADS-B

8.3 Modeling and Verifying ADS-B and its extensions

By modeling ADS-B's communication system in the GSD framework, we specify the system in a language that is more formal than the written English used by the system designers and more abstract than the specification languages that are used as input to the different verification tools. That helps closing the gap between the two and makes it easier to provide feedback to the system designers. This way, it enables not only the formal verification of the ADS-B system but also makes it easier to report the results to the system designers.

8.3.1 Modeling ADS-B

We started by reviewing the existing American ADS-B standards [RTC02, RTC11a, RTC11b] and other documents that describe ADS-B [FAAa, Avi12] in order to identify ADS-B's communication protocol. A simple overview of the identified system is shown in Figure 8.3.

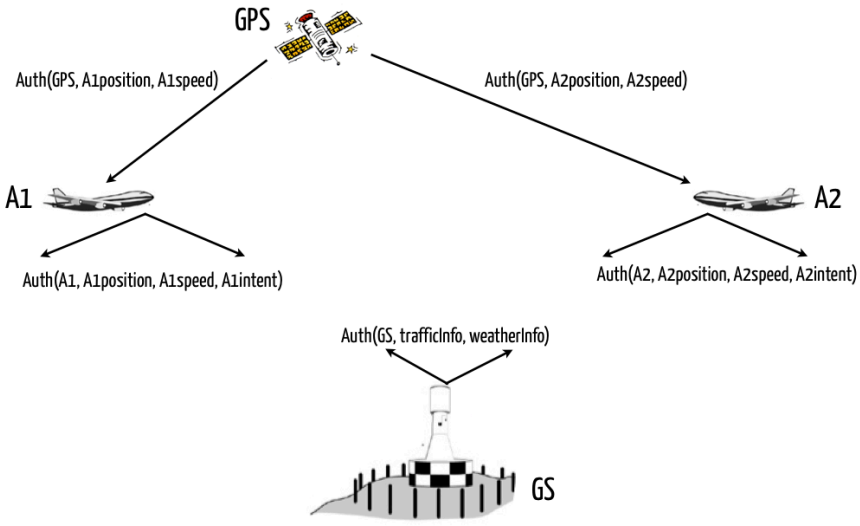


Figure 8.3: Overview of the ADS-B model

The ADS-B system is composed by three principal types: the aircraft, the ground stations and the satellites. We made some abstractions in our model that are worth mentioning. We abstracted the details regarding the data being transmitted in the ADS-B broadcasts and we also abstracted the way the aircraft acquire their position and speed by having a principal called Satellite sending that information to the aircraft. These are abstractions that make the modeling simpler but that still allow the analysis that we want to perform. Besides those messages, we also modeled the ADS-B broadcasts made both by the ground stations and the aircraft.

As mentioned above, we modeled the broadcast of messages buy specifying that a principal sends a message to *Air*. This way of modeling the system, together with the BAK tool allow us to easily model Denial of Service and also, using the modular attacker capabilities of the BAK tools, when a principal is out of reach.

We were mostly interested in analyzing the security properties of the exchanged

messages and to reason about that, it was enough to model a small part of the system when each of the principals are sending one message. That system model can be seen in Listing 8.1.

```
Satellite → A1 : Satellite, position1, speed1;
Satellite → A2 : Satellite, position2, speed2;
A1 → Air : A1, position1, speed1, intent1;
A2 → Air : A2, position2, speed2, intent2;
GS → Air : GS, trafficInfo, weatherInfo;
```

Listing 8.1: ADS-B Model

8.3.2 Validation of the ADS-B Model

There were mainly two processes we used to to perform the validation of our model. We presented our model to the experts at NASA and discussed it with them, analyzing their feedback and we also modeled and verified McCallie's attack taxonomies and compared the results with the descriptions given by McCallie [MBM11]. This verification is presented bellow.

8.3.2.1 McCallie's attack taxonomies

McCallie [MBM11] identifies different ADS-B attack taxonomies that we believe are important to consider when performing the validation of our model of the ADS-B communication system. All these attacks were successfully verified with BAK tool against the modeled ADS-B communication system. The different attacks are:

Aircraft Reconnaissance

This attack consists on intercepting and decoding ADS-B transmissions by exploring its lack of confidentiality. This enables any outsider with an ADS-B enable receiver to acquire the transmitted flight information, such as the aircraft's ID, together with its position, speed and intent.

This attack consists on intercepting and decoding ADS-B transmissions. And, since it is a passive attack, it can just be modeled by testing if the attacker is able to see the speed and location information sent by the aircraft. That can be done with a goal such as `attacker sees position1`, where `position1` is the

position broadcasted by one of the aircraft.

Ground Station Flood Denial

This attack will disrupt transmission at the ground station, preventing it from receiving the ADS-B transmissions from the aircraft. This can be done by using a jamming device next to the ground station.

This attack will disrupt transmission at the ground station and is, therefore, an active attack, more complex to model. One way this attack can be interpreted is that, if an interference successfully jams a ground station, then it is changing the topology of our network model. This can then be modeled by changing the way messages are exchanged in the system, i.e., when the aircraft broadcast their information, the ground station being jammed will not see those messages. In our model, that is done by changing one of the core rules. The original rule expresses that if a `message` is sent to `Air`, the former is seen by every principal in the system. By changing the rule so that the jammed ground station does not see the messages sent to `Air`, then we are modeling the outcomes of a successful jamming attack on that ground station.

Ground Station Target Ghost Inject

This attack explores the lack of secure authentication mechanisms by injecting an ADS-B signal into a ground station with the intent of leading the ground station to believe that a fake aircraft is in the airspace.

The importance of this attack is dismissed by FAA by saying that the integration with a secondary navigation system, such as the radar, will identify these ghost aircraft. FAA is, therefore, arguing for the security of ADS-B using its not so well defined integration of ADS-B with other navigation systems. We believe it would be more desirable to have this security embedded in ADS-B to start with.

This active attack injects an ADS-B signal into a ground station and is modeled by having the attacker broadcast false information regarding the position and speed of an aircraft.

Aircraft Flood Denial

This is similar to the ground station flood denial attack, except that the target

is an aircraft. This attack is harder to perform, when compared to the ground station flood denial, specially when the aircraft is airborne. A situation where this attack would be more feasible is next to airport, where aircraft are taking off and landing.

This is similar to the ground station flood denial attack and is modeled in a similar fashion. The difference being, in this case, that the aircraft is the one who is not able to see the messages broadcasted by the other principals.

Aircraft Target Ghost Inject

Similar to the ground station target ghost inject attack, except that the target is an aircraft. This makes the attack potentially more dangerous since the aircraft will not be able to directly check the received position of the ghost aircraft. This can, at least, cause momentary panic with the pilots who might think that the ghost aircraft is dangerously close to their aircraft.

This attack is modeled in the same way the ground station target ghost inject attack is modeled. That is because whenever an attacker broadcasts the false information of the ghost aircraft, it will also be received by the aircraft that are in receiving range.

Ground Station Multiple Ghost Inject

This attack injects several ADS-B signals into a ground station. It is similar to the ground station target ghost inject attack, except that multiple targets are injected into the system. This can be used to overwhelm the surveillance system and create confusion and eventually make the user lose trust in the system.

This active attack injects several ADS-B signals into a ground station and is modeled by using several messages similar to the one modeled in Ground Station Target Ghost Inject.

8.3.3 Valovage's ADS-B Security Extensions

Valovage et al. have been working on extending ADS-B with mechanisms that enable authentication and/or confidentiality of the ADS-B broadcast messages [Val06, VV+07].

Valovage's suggestion uses secret keys shared between the principals present in the system (ideally there are different keys for each pair of principals). These keys, used together with Hashing and MAC, functions enable the authentication of the principals while adding a small number of extra necessary bits. As for confidentiality, Valovage also suggests the use of shared-key cryptography, but performing data encryption in order to achieve confidentiality, which will will require more extra bits.

A positive aspect of his approach is that it is retro compatible, i.e., compatible with the present implementation, when just using the extra authentication. It is not possible to be retro compatible when using encryption to achieve confidentiality, since in this case no information in plain-text is sent and, therefore, only the principals using this scheme and with access to the shared-keys will be able to decrypt the sent message.

In the suggested solution, present in the patent [VV+07] and shown in Table 8.1, different combinations of authentication and confidentiality are proposed. Furthermore, Valovage also presents options that provide strong authentication by having freshness guarantees present when performing the authentication, which as we previously discussed prevents replay attacks. All these solutions were successfully verified with the GSD framework using the BAK tool.

Confidentiality / Authentication	No	Yes	Yes (and Fresh)
No	Current ADS-B	Section 8.3.3.1	Section 8.3.3.3
Yes	—	Section 8.3.3.2	Section 8.3.3.4

Table 8.1: Different options on Valovage's suggested ADS-B extension

8.3.3.1 ADS-B with Authentication

This is shown in Fig. 2 of the patent. The principal that is being authenticated sends not only the normal ADS-B information but also some extra bits that result from applying the MAC function to the Hash of a default value, the principal's Id, the ADS-B data and a secret key that is shared between the principal and the intended message recipient. This way, when the intended

recipient receives this extra bits, together with the data sent in plain-text, he is able to authenticate the data. This can be done because the principal that receives the messages has all the data (the default number, the other principal's Id, the sent ADS-B data and the secret key) and, therefore, is able to feed them into the Hash and MAC functions and compare the resulting bits with the extra ones received in the ADS-B message. The modeled message exchange is shown in Listing 8.2.

```
A1 -> GS: Hash&Mac(default,
                   Id(A1),
                   adsbData,
                   ShKey(GS, A1)),
          Id(A1),
          adsbData
```

Listing 8.2: ABS-B with Authentication

One thing worth noticing is that this solution does not provide any guarantees of freshness to the principal receiving the message, so it is subject to replay attacks and it is not, therefore, suitable for using in very secure sensitive areas. However, it is a very lightweight solution that adds some security to ADS-B and that could be used when some message authentication is required, but no high concerns regarding potential replay attacks are present. Furthermore, any principal external to the system is still able to read the information sent in plain-text, i.e., the Id of the principal together with the ADS-B data.

8.3.3.2 ADS-B with Authentication and Confidentiality

This extension, shown in Fig. 3 of the patent, is similar to the one presented in the previous section, but both the Id of the principal and the ADS-B data are being encrypted with the secret key prior to being sent and also prior to being fed into the Hash function. This requires the principal receiving the principal to decrypt the data with the secret key. This extends the previous suggestion with confidentiality, which prevents any outside principal to read the sent information. This solution is modeled as shown in Listing 8.3.

```
A1 -> GS: Hash&Mac(default,
                   Enc((Id(A1), data), ShKey(GS,A1)),
                   ShKey(GS, A1)),
          Enc(Id(A1), data, ShKey(GS,A1))
```

Listing 8.3: ABS-B with Authentication and Confidentiality

8.3.3.3 ADS-B with Fresh Authentication

This proposed extension of ADS-B not only provides authentication to the principal sending the ADS-B message but also fresh authentication, which will protect the system against the possibility of replay attacks. This, as one could expect, comes at a price, in this case network bandwidth. Similarly to the implementation of the strong authentication shown in Section 4.2.2, Valovage implements freshness by having the authenticator — in the ADS-B's case, the Ground Station — broadcasting a fresh challenge that will be used by the principal being authenticating. When performing the hashing of the different values, the principal also uses the broadcasted fresh challenge. This way, the Ground Station has guarantees that the information was recently broadcasted by the authenticated principal. Using this mode will still be backwards compatible. The model for this extension is shown in Listing 8.4.

```
GS -> A1: nonce
      A1 -> GS: Hash&Mac(nonce,
                         Id(A1),
                         adsbData,
                         ShKey(GS,A1)),
                Id(A1),
                adsbData
```

Listing 8.4: ABS-B with Fresh Authentication

8.3.3.4 ADS-B with Fresh Authentication and Confidentiality

This extends the previous suggestion with confidentiality. It is the most secure of the suggested extensions to ADS-B since it provides authentication with freshness guarantees together with confidentiality, which is, in fact, the security properties given by the strong security module present in the Abstract Global level of the GSD framework. This extension's model is presented in Listing 8.5.

```
GS -> A1: nonce
A1 -> GS: Hash&Mac(nonce,
                   Enc((Id(A1), adsbData), shKey(GS,A1)),
                   ShKey(GS,A1)),
          Enc((Id(A1), adsbData), ShKey(GS,A1))
```

Listing 8.5: ABS-B with Fresh Authentication and Confidentiality

8.4 Suggested extensions to ADS-B

In this section, we present some ideas on how to further extend ADS-B with security capabilities. The goal of this section is, therefore, to show a range of different security mechanisms that can be used. Each have different pros and cons and the choice should be made by having those in consideration.

The biggest challenge regarding these suggested extensions is the key-management. Some ideas are given on how to perform it, but further work needs to be done in order to arrive at an optimal solution.

8.4.1 Shared Key Cryptography - key per airspace

In this approach, a shared-key would be used for authentication and confidentiality within an airspace, which could be a country, or a state, or the oceanic airspace. This means that all the aircraft that fly through a specific airspace will have the respective shared-key. The distribution of that key could be done when the aircraft is getting authorized for take off. Since its route has to be authorized, it could also receive (securely) from the Ground System the shared-keys of the different airspaces that the aircraft is flying through. Then, when flying in one airspace, it would use the respective shared key when the need for authentication and/or confidentiality arises by, respectively, using the shared key in the hash function or encrypting the message with the shared key, as suggested by Valovage et al.

This approach requires trusting in a big number of principals and it would only provide a certain degree of authentication and/or confidentiality, i.e., using this system, an authorized principal would be able to be authenticated and read a confidential message sent from another authorized principal in that airspace, but the level of assurance that the message was sent from a specific aircraft is not as high as when compared with the suggestion bellow. This is the case because a high number of principals will have access to the shared-key. We would argue that, despite that, it would still provide ADS-B with a better security level than the present one, since it makes it impossible for a non-authorized principal to authenticate or read confidential ADS-B messages, assuming the shared-key is leaked.

It would, however, be fairly complicate to manage all the keys, specially in case one them is leaked. If a non-authorized principal has access to a shared-key, the ground system would have to broadcast a key update for that airspace — via ADS-B or eventually ADS-C, in case it is an oceanic airspace — but it is hard

to provide guarantees that all the aircraft will have the updated key before the non-authorized principal can explore its knowledge of the leaked shared-key.

8.4.2 Public Keys Cryptography

In this suggestion, a Public-Key Infrastructure would be used to extend ADS-B with higher security assurances. Each aircraft has a public-/private-key pair and the ground system also has a public-/private-key pair.

Regarding the aircraft key pair, as it is common in a PKI, only the aircraft know their own private-key of its pair while the public-key is accessible to all the principals in the system. The ground system would act as the PKI certificate authority (CA) by maintaining a database of all the aircraft public-keys and in case an aircraft needs a public-key to authenticate or read a confidential message that it received, it can send a request to the ground system, which will then transmit the required public-key signed with its own private key. This request and reply can either happen via ADS-B in case a ADS-B ground stations is within of the aircraft, or if that is not the case (which will not happen that often) the request can be made via satellite, for example using ADS-C. The satellite communication (which ADS-C uses) is not to be used often, but we believe that most of the times there will be an ADS-B station within reach and, therefore, ADS-B can be used.

This system enables both authentication and confidentiality. The former by encrypting an hash of the ADS-B message with ones private-key and the latter by encrypting the whole message with the intended recipient's public-key. We are aware that using confidentiality for air-to-air communication in an area with a high density of aircraft will not be simple with this suggestion, since one would have to encrypt and send the message as many times as the number of aircraft within reach. If confidentiality becomes really important, it could be advantageous to consider an hybrid solution, with a public-key system for authentication and a shared-key system similar to the one presented in the previous section for confidentiality.

Chapter 9

Conclusion

> The only truly secure system is one that is powered off, cast in a block of concrete, and sealed in a lead-lined room with armed guards.
>
> Eugene Spafford

In today's interconnected world, it is frequently impossible to adopt the solution mentioned by Eugene Spafford and that is why it is so important to build communication systems that are secure. In this thesis, I presented a framework that targets precisely that. The Guided System Development (GSD) framework aims at helping the modeling, verification, and implementation of secure communication systems. The framework is composed by three different abstraction levels: the Abstract Global (Chapter 4), the Concrete Global (Chapter 5), and the Concrete Endpoint (Chapter 6). The specification of the communication system is modeled in the Abstract Global level and step-wise is used to translate that specification to the other levels of the tool: from the Abstract Global level to the Concrete Global level by replacing the security modules with their implementation and from the Concrete Global level to the Concrete Endpoint level by performing an endpoint projection of the specification, transforming it from a global view of the communication system into an endpoint view of the same communication system.

The modeling of the communication system is performed with a simple and intuitive language that has the necessary constructs (security modules) for modeling the security requirements of the exchanged messages. This modeling phase is the main focus of the Abstract Global level of the framework.

One of the main values of the GSD framework is the connection with different tools that used formal methods to perform security verification of communication systems. Having the connection to these different tools allows for an extensive analysis of the communication system and its security properties. Each verification tools has its strong points and connecting to a number of tools allows for a more extensive analysis of the modeled system. The developed GSD framework prototype connects to OFMC, LySatool, and the BAK tool, which was developed as part of this work specifically for the GSD framework and is presented in Chapter 3. The connection to LySatool is performed from the Concrete Endpoint level and is presented in Section 6.4.1, while the connection to OFMC is performed both in the Abstract Global level and in the Concrete Global level, presented in Section 4.3.1 and Section 5.3.2, respectively. The GSD framework prototype also provides the automatic translation of the communication skeleton for each of the modeled principals to Java code.

9.1 Future Work

In this Section, I discuss future work possibilities regarding both the research side of my work and also the side more related with the prototype development.

There are some interesting possibilities in terms of future work related with the research presented here: one possibility would be to extend the Abstract Global level with more security modules that would provide different communication properties, such as non-repudiation and message ordering, and also define their implementation. Another interesting possibility would be to enable the description of receiver actions in the Abstract Global level specification using, for example, extra notations. One could also further explore the analysis of scenarios where the modular attacker property of the BAK tool is required, which would happen, for example, in real cyber-physical systems where the attacker is limited geographically. Another possibility would be to add the connection to other verification tools: for example, some that provide quantitative security analysis. Another interesting research direction would be to provide more assurances regarding the generated code, this could either be done by formally proving the correctness of the translation from a verified model or by extracting a verifiable specification from the generated code. Finally, other use cases of the application of the GSD framework could also be studied.

There are some details on the prototype implementation of the GSD framework that were not finalized due to them being predominantly engineering problems and my main goal was to focus on the research possibilities of this work. Having said that, if one wants to improve the prototype tool, there are a number of de-

tails one could focus on. The BAK tool could be extended in order to enable the analysis and interpretation of the satisfiability model given as output — when the analysis result is SAT — in order to provide more useful feedback to the system designer. Regarding the translation to AnB code, more information could automatically generated, such as the initial knowledge for the principals and the modeled dummy messages in between the modeled messages. Furthermore, one could also explore the OFMC analysis with dishonest principals, which could be interesting for the authentication cases. There are also some possibilities regarding the translation to LySa and the fact that the missing information could be automatically generated: one piece of missing information in the translation is the description of the cryptopoints — which are explained in Section 6.4.1 — and the other would be the declaration of the sent messages as new, i.e., modeling that they are created by the sender so that the confidentiality analysis result is correct. Regarding the translation to Java: more information could be extracted, but more importantly, the translation could be improved to generate fully compilable Java programs instead of the communication skeletons that are generated in the current prototype implementation. Finally, one should also consider adding more plugins to the GSD framework and, therefore, more possible implementations of the security modules.

Appendix A

Code

A.1 Java Library

Some of the functions defined in the used Java Library are shown below.

```
public static String generateNonce(){
  String result = null;
  SecureRandom random;
  try {
    random = SecureRandom.getInstance("SHA1PRNG");
    byte nonce[] = new byte[20];
    random.nextBytes(nonce); // compute a 20 byte random nonce
    result = new String(nonce);
  } catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
  }
  return result;
}
```

Listing A.1: Generate unique string.

```
static String pubEncryption(String msg, String key){
  String result = new String();
  try {
    FileInputStream keyfis = new FileInputStream("keys/" + key + ".
        pubkey");
    byte[] encKey = new byte[keyfis.available()];
    keyfis.read(encKey);
    keyfis.close();

    X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encKey);

    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

    // Instantiate the cipher
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE, pubKey);
    byte[] encrypted = cipher.doFinal(msg.getBytes());

    // converts to base64
    result = new String(Base64.encodeBase64(encrypted));

  } catch (IOException e) {
    e.printStackTrace();
  } catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
  } catch (NoSuchPaddingException e) {
    e.printStackTrace();
  } catch (InvalidKeyException e) {
    e.printStackTrace();
  } catch (IllegalBlockSizeException e) {
    e.printStackTrace();
  } catch (BadPaddingException e) {
    e.printStackTrace();
  } catch (InvalidKeySpecException e) {
    e.printStackTrace();
  }
  return result;
}
```

Listing A.2: Public-Key Encryption.

```
static String pubDecryption(String msg, String key){
  String result = new String();
  try {
    FileInputStream keyfis = new FileInputStream("keys/" + key + ".
        privkey");
    byte[] encKey = new byte[keyfis.available()];
    keyfis.read(encKey);
    keyfis.close();

    PKCS8EncodedKeySpec privKeySpec = new PKCS8EncodedKeySpec(
        encKey);

    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    PrivateKey privKey = keyFactory.generatePrivate(privKeySpec);

    // Instantiate the cipher
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.DECRYPT_MODE, privKey);

    //decode message
    byte[] decodedText = Base64.decodeBase64(msg.getBytes());
    byte[] original = cipher.doFinal(decodedText);
    result = new String(original);

  } catch (IOException e) {
    e.printStackTrace();
  } catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
  } catch (NoSuchPaddingException e) {
    e.printStackTrace();
  } catch (InvalidKeyException e) {
    e.printStackTrace();
  } catch (IllegalBlockSizeException e) {
    e.printStackTrace();
  } catch (BadPaddingException e) {
    e.printStackTrace();
  } catch (InvalidKeySpecException e) {
    e.printStackTrace();
  }
  return result;
}
```

Listing A.3: Public-Key Decryption.

```
static String shEncryption(String msg, Arg key){
  byte[] raw = null;
  String result = new String();

  if(key.getData().length() < 10) {
    try {
      FileInputStream keyfis = new FileInputStream("keys/" + (key.
          getData()) + ".shkey");
      raw = new byte[keyfis.available()];
      keyfis.read(raw);
      keyfis.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  else {
    raw = Base64.decodeBase64(key.getData().getBytes());
  }

  try{
    SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");

    // Instantiate the cipher
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
    byte[] encrypted = cipher.doFinal(msg.getBytes());

    // converts to base64
    result = new String(Base64.encodeBase64(encrypted));

  } catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
  } catch (NoSuchPaddingException e) {
    e.printStackTrace();
  } catch (InvalidKeyException e) {
    e.printStackTrace();
  } catch (IllegalBlockSizeException e) {
    e.printStackTrace();
  } catch (BadPaddingException e) {
    e.printStackTrace();
  }
  return result;
}
```

Listing A.4: Shared-Key Encryption.

```
static String shDecryption(String msg, Arg key){
  byte[] raw = null;
  String result = new String();

  if(key.getData().length() < 10) {
    try {
      FileInputStream keyfis = new FileInputStream("keys/" + (key.
          getData()) + ".shkey");
      raw = new byte[keyfis.available()];
      keyfis.read(raw);
      keyfis.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  else {
    raw = Base64.decodeBase64(key.getData().getBytes());
  }

  try{
    SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");

    // Instantiate the cipher
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, skeySpec);

    //decode message
    byte[] decodedText = Base64.decodeBase64(msg.getBytes());
    byte[] original = cipher.doFinal(decodedText);
    result = new String(original);

  } catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
  } catch (NoSuchPaddingException e) {
    e.printStackTrace();
  } catch (InvalidKeyException e) {
    e.printStackTrace();
  } catch (IllegalBlockSizeException e) {
    e.printStackTrace();
  } catch (BadPaddingException e) {
    e.printStackTrace();
  }
  return result;
}
```

Listing A.5: Shared-Key Decryption.

Bibliography

[AF06] Pedro Adão and Cédric Fournet. Cryptographically sound implementations for communicating processes. In *ICALP (2)*, pages 83–94, 2006.

[AFG02] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. *Inf. Comput.*, 174(1):37–83, 2002.

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[AVA11] AVANTSSAR. The avantssar project. http://www.avantssar.eu, 2011.

[AVI03] AVISPA. Deliverable 2.3: The intermediate format. http://www.avispa-project.org, 2003.

[Avi12] Duncan Avionics. *Straight Talk About ADS-B*. Duncan Avionics, 2012.

[BAN90] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8:18–36, February 1990.

[BBD+05] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H.R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[BBDNL08] Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Sessions and pipelines for structured service programming. In Gilles Barthe and Frank de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer Berlin / Heidelberg, 2008.

[BDGH97] JP Bekmann, P. De Goede, and ACM Hutchison. SPEAR: Security protocol engineering and analysis resources. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, pages 3–5, 1997.

[BF08] Michele Bugliesi and Riccardo Focardi. Language based secure communication. In *CSF*, pages 3–16, 2008.

[BF10] Michele Bugliesi and Riccardo Focardi. Channel abstractions for network security. *Mathematical Structures in Computer Science*, 20(1):3–44, 2010.

[BFGP04] Karthikeyan Bhargavan, Cédric Fournet, AndrewD. Gordon, and Riccardo Pucella. Tulafale: A security tool for web services. In FrankS. Boer, MarcelloM. Bonsangue, Susanne Graf, and Willem-Paul Roever, editors, *Formal Methods for Components and Objects*, volume 3188 of *Lecture Notes in Computer Science*, pages 197–222. Springer Berlin Heidelberg, 2004.

[BFGT08] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1):5, 2008.

[BG07] Michele Bugliesi and Marco Giunti. Secure implementations of typed channel abstractions. In *POPL*, pages 251–262, 2007.

[Bla02] Bruno Blanchet. From secrecy to authenticity in security protocols. In ManuelV. Hermenegildo and Germán Puebla, editors, *Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 342–359. Springer Berlin Heidelberg, 2002.

[Bla08] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *Dependable and Secure Computing, IEEE Transactions on*, 5(4):193–207, 2008.

[Bla09] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 01 2009.

[BM93] Colin Boyd and Wenbo Mao. On a limitation of ban logic. In *EUROCRYPT*, pages 240–247, 1993.

[BM11] Michele Bugliesi and Paolo Modesti. AnBx - Security Protocols Design and Verification. In Alessandro Armando and Gavin Lowe, editors, *Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, volume 6186 of *Lecture Notes in Computer Science*, pages 164–184. Springer Berlin Heidelberg, 2011.

[Buc05] Mikael Buchholtz. *User's Guide for the LySatool version 2.01*. DTU, April 2005.

[CB12] David Cade and Bruno Blanchet. From computationally-proved protocol specifications to implementations. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 65–74. IEEE, 2012.

[CHL+00] Charles T. Cullen, Peter B. Hirtle, David Levy, Clifford A. Lynch, Jeff Rothenberg, and Charles T. Cullen Is President. Authenticity in a digital environment, 2000.

[CHY07] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *Programming Languages and Systems*, pages 2–17. Springer, 2007.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DLMS04] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 01 2004.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[DNFP98] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24:315–330, 1998.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[DR08] T. Dierks and E. Rescorla. RFC 5246 - the transport layer security (TLS) protocol version 1.2. Technical report, August 2008.

[DY83] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[FAAa] FAA. ADS-B General Information. http://www.faa.gov/nextgen/implementation/portfolio/trans_support_progs/adsb/general/.

[FAAb] FAA. What Is NextGen? http://www.faa.gov/nextgen/why_nextgen_matters/what.

[FG00] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *APPSEM*, pages 268–332, 2000.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.

[GNN09] H. Gao, F. Nielson, and H.R. Nielson. Protocol Stacks for Services. In *Foundations of computer security*, 2009.

[Gol96] D. Gollmann. What do we mean by entity authentication? In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 46–54, 1996.

[HS99] ACM Hutchison and E. Saul. SPEAR II: The Security Protocol Engineering and Analysis Resource. In *2d Annual South African Telecomm., Networks, and Applications Conference, Durban, South Africa*, pages 171–177, 1999.

[KS05] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.

[Lie93] Armin Liebl. Authentication in distributed systems: a bibliography. *SIGOPS Oper. Syst. Rev.*, 27:31–41, October 1993.

[Low97] Gavin Lowe. A hierarchy of authentication specifications. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 31–43, 1997.

[LVH02] S. Lukell, C. Veldman, and A. Hutchison. Automated attack analysis and code generation in a unified, multi-dimensional security protocol engineering framework. *Comp. Science Hon*, 2002.

[MÖ9] Sebastian Mödersheim. Algebraic Properties in Alice and Bob Notation. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pages 433 –440, 2009.

[MBM11] Donald McCallie, Jonathan Butts, and Robert Mills. Security analysis of the ADS-B implementation in the next generation air transportation system. *International Journal of Critical Infrastructure Protection*, 4(2):78 – 87, 2011.

[Mod11] Paolo Modesti. *Verified Security Protocol Modeling and Implementation with AnBx*. PhD thesis, Università Ca' Foscari di Venezia, 2011.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41 – 77, 1992.

[MS96] Ueli M. Maurer and Pierre E. Schmid. A calculus for security bootstrapping in distributed systems. *Journal of Computer Security*, 4(1):55–80, 1996.

[MV09a] Sebastian Mödersheim and Luca Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 166–194. Springer Berlin / Heidelberg, 2009.

[MV09b] Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels. In *ESORICS*, pages 337–354, 2009.

[MV09c] Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels (extended version). T. Rep. RZ3724, IBM Zurich Research Lab, http://domino.research.ibm.com/library/cyberdig.nsf, 2009.

[MZ09] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, August 2009.

[NKHBM04] Anthony Nadalin, Chris Kaler, Phillip Hallam-Baker, and Ronald Monzillo. Web services security: Soap message security 1.0 (ws-security 2004). Technical report, OASIS, 2004.

[NNS+04] F. Nielson, H.R. Nielson, Hongyan Sun, M. Buchholtz, R.R. Hansen, H. Pilegaard, and H. Seidl. The succinct solver suite. *Tools and Algorithms for the Construction and Analysis of Systems. 10th International Conference, TACAS 2004. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004. Proceedings (Lecture Notes in Computer Science Vol.2988)*, pages 251–265, 2004.

[Par07] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[Par09] T. Parr. Stringtemplate documentation. http://www.antlr.org/wiki/display/ST/StringTemplate+Documentation, May 2009.

[QR] Jose Quaresma and Kristin Y. Rozier. Modeling, Analyzing and Extending the ADS-B communication system. to be submitted (available by request).

[Qua10] Jose Quaresma. A protocol implementation generator. Master's thesis, Technical University of Denmark (DTU), Kgs. Lyngby, Denmark, June 2010.

[RS60] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[RTC02] RTCA. DO-242A: Minimum Aviation System Performance Standards for Automatic Dependent Surveillance Broadcast (ADS-B). Technical report, RTCA, 2002.

[RTC11a] RTCA. DO-260B: Minimum Operational Performance Standards (MOPS) for 1090 MHz Extended Squitter Automatic Dependent Surveillance – Broadcast (ADS-B) and Traffic Information Services – Broadcast (TIS-B). Technical report, RTCA, 2011.

[RTC11b] RTCA. DO-282B: Minimum Operational Performance Standards (MOPS) for Universal Access Transceiver (UAT) Automatic Dependent Surveillance - Broadcast (ADS-B). Technical report, RTCA, 2011.

[SBP01] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1):47–74, 2001.

[SCF+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM.

[Sel03] Peter Selinger. Models for an adversary-centric protocol logic. *Electronic Notes in Theoretical Computer Science*, 55(1):69–84, 2003.

[sen10] SENSORIA: Software Engineering for Service-Oriented Overlay Computers. http://www.sensoria-ist.eu, 2010.

[SPP01] Dawn Song, Adrian Perrig, and Doantam Phan. Agvi—automatic generation, verification, and implementation of security protocols. In *Computer Aided Verification*, pages 241–245. Springer, 2001.

[Syv91] P. Syverson. The use of logic in the analysis of cryptographic protocols. In *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pages 156 –170, may 1991.

[Uni11] Internation Telecommunication Union. Asn.1. https://www.itu.int/ITU-T/asn1/introduction/index.htm, 2011.

[Val06] E. Valovage. Enhanced ADS-B Research. In *25th Digital Avionics Systems Conference, 2006 IEEE/AIAA*, pages 1 –7, oct. 2006.

[VV+07] M. Viggiano, E. Valovage, et al. Secure ADS-B Authentication System And Method, October 12 2007. WO Patent 2,007,115,246.

[W3C05] W3C. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10/, November 2005.

[Wei99] Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Automated Deduction — CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 314–328. Springer Berlin Heidelberg, 1999.

[Wic94] Stephen B. Wicker. *Reed-Solomon Codes and Their Applications*. IEEE Press, Piscataway, NJ, USA, 1994.